Brian 2 Documentation

Release 2.5.0.3

Brian authors

Jan 27, 2022

CONTENTS

Brian is a simulator for spiking neural networks. It is written in the Python programming language and is available on almost all platforms. We believe that a simulator should not only save the time of processors, but also the time of scientists. Brian is therefore designed to be easy to learn and use, highly flexible and easily extensible.

To get an idea of what writing a simulation in Brian looks like, take a look at *a simple example*, or run our interactive demo.

Once you have a feel for what is involved in using Brian, we recommend you start by following the *installation instructions*, and in case you are new to the Python programming language, having a look at *Running Brian scripts*. Then, go through the *tutorials*, and finally read the *User Guide*.

While reading the documentation, you will see the names of certain functions and classes are highlighted links (e.g. `PoissonGroup`). Clicking on these will take you to the "reference documentation". This section is automatically generated from the code, and includes complete and very detailed information, so for new users we recommend sticking to the *User's guide*. However, there is one feature that may be useful for all users. If you click on, for example, `PoissonGroup`, and scroll down to the bottom, you'll get a list of all the example code that uses `PoissonGroup`. This is available for each class or method, and can be helpful in understanding how a feature works.

Finally, if you're having problems, please do let us know at our *support page*.

Please note that all interactions (e.g. via the mailing list or on github) should adhere to our *Code of Conduct*.

Contents:

ONE

INTRODUCTION

1.1 Installation

- *Standard install*

- *Updating an existing installation*

- *Requirements for C++ code generation*

- *Development install*

- *Installing other useful packages*

- *Testing Brian*

There are various ways to install Brian, and we recommend that you chose the installation method that they are most familiar with and use for other Python packages. If you do not yet have Python installed on your system (in particular on Windows machines), you can install Python and all of Brian's dependencies via the Anaconda distribution. You can then install Brian with the `conda` package manager as detailed below.

Note: You need to have access to Python >=3.7 (see Brian's *support policy*). In particular, Brian no longer supports Python 2 (the last version to support Python 2 was *Brian 2.3*). All provided Python packages also require a 64 bit system, but every desktop or laptop machine built in the last 10 years (and even most older machines) is 64 bit compatible.

If you are relying on Python packages for several, independent projects, we recommend that you make use of separate environments for each project. In this way, you can safely update and install packages for one of your projects without affecting the others. Both, `conda` and `pip` support installation in environments – for more explanations see the respective instructions below.

1.1.1 Standard install

conda package

PyPI package (`pip`)

Ubuntu/Debian package

Fedora package

We recommend installing Brian into a separate environment, see conda's documentation for more details. Brian 2 is not part of the main Anaconda distribution, but built using the community-maintained conda-forge project. You will therefore have to to install it from the conda-forge channel. To do so, use:

```
conda install -c conda-forge brian2
```

You can also permanently add the channel to your list of channels:

```
conda config --add channels conda-forge
```

This has only to be done once. After that, you can install and update the brian2 packages as any other Anaconda package:

```
conda install brian2
```

We recommend installing Brian into a separate "virtual environment", see the Python Packaging User Guide for more information. Brian is included in the PyPI package index: https://pypi.python.org/pypi/Brian2 You can therefore install it with the `pip` utility:

```
python -m pip install brian2
```

In rare cases where your current environment does not have access to the `pip` utility, you first have to install `pip` via:

```
python -m ensurepip
```

If you are using a recent Debian-based Linux distribution (Debian itself, or one if its derivatives like Ubuntu or Linux Mint), you can install Brian using its built-in package manager:

```
sudo apt install python3-brian
```

Brian releases get packaged by the Debian Med team, but note that it might take a while until the most recent version shows up in the repository.

If you are using Fedora Linux, you can install Brian using its built-in package manager:

```
sudo dnf install python-brian2
```

Brian releases get packaged by the NeuroFedora team, but note that it might take a while until the most recent version shows up in the repository.

1.1.2 Updating an existing installation

How to update Brian to a new version depends on the installation method you used previously. Typically, you can run the same command that you used for installation (sometimes with an additional option to enforce an upgrade, if available):

conda package

PyPI package (`pip`)

Ubuntu/Debian package

Fedora package

Depending on whether you added the `conda-forge` channel to the list of channels or not (see above), you either have to include it in the update command again or can leave it away. I.e. use:

```
conda update -c conda-forge brian2
```

if you did not add the channel, or:

```
conda update brian2
```

if you did.

Use the install command together with the `--upgrade` or `-U` option:

```
python -m pip install -U brian2
```

Update the package repository and ask for an install. Note that the package will also be updated automatically with commands like `sudo apt full-upgrade`:

```
sudo apt update
sudo apt install python3-brian
```

Update the package repository (not necessary in general, since it will be updated regularly without asking for it), and ask for an update. Note that the package will also be updated automatically with commands like `sudo dnf upgrade`:

```
sudo dnf check-update python-brian2
sudo dnf upgrade python-brian2
```

1.1.3 Requirements for C++ code generation

C++ code generation is highly recommended since it can drastically increase the speed of simulations (see *Computational methods and efficiency* for details). To use it, you need a C++ compiler and Cython (automatically installed as a dependency of Brian).

Linux and OS X

Windows

On Linux and Mac OS X, the conda package will automatically install a C++ compiler. But even if you install Brian in a different way, you will most likely already have a working C++ compiler installed on your system (try calling `g++ --version` in a terminal). If not, use your distribution's package manager to install a `g++` package.

On Windows, *Runtime code generation* (i.e. Cython) requires the Visual Studio compiler, but you do not need a full Visual Studio installation, installing the much smaller "Build Tools" package is sufficient:

- Install the Microsoft Build Tools for Visual Studio.

- In Build tools, install C++ build tools and ensure the latest versions of MSVCv… build tools and Windows 10 SDK are checked.

- Make sure that your `setuptools` package has at least version 34.4.0 (use `conda update setuptools` when using Anaconda, or `python -m pip install --upgrade setuptools` when using pip).

For *Standalone code generation*, you can either use the compiler installed above or any other version of Visual Studio.

Try running the test suite (see *Installing other useful packages* below) after the installation to make sure everything is working as expected.

1.1.4 Development install

When you encounter a problem in Brian, we will sometimes ask you to install Brian's latest development version, which includes changes that were included after its last release.

We regularly upload the latest development version of Brian to PyPI's test server. You can install it via:

```
python -m pip install --upgrade --pre -i https://test.pypi.org/simple/ Brian2
```

Note that this requires that you already have all of Brian's dependencies installed.

If you have `git` installed, you can also install directly from github:

```
python -m pip install git+https://github.com/brian-team/brian2.git
```

Finally, in particular if you want to either contribute to Brian's development or regularly test its latest development version, you can directly clone the git repository at github (https://github.com/brian-team/brian2) and then run `pip install -e .`, to install Brian in "development mode". With this installation, updating the git repository is in general enough to keep up with changes in the code, i.e. it is not necessary to install it again.

1.1.5 Installing other useful packages

There are various packages that are useful but not necessary for working with Brian. These include: matplotlib (for plotting), pytest (for running the test suite), ipython and jupyter-notebook (for an interactive console).

conda package

PyPI package (`pip`)

```
conda install matplotlib pytest ipython notebook
```

```
python -m pip install matplotlib pytest ipython notebook
```

You should also have a look at the brian2tools package, which contains several useful functions to visualize Brian 2 simulations and recordings.

conda package

PyPI package (`pip`)

As of now, `brian2tools` is not yet included in the `conda-forge` channel, you therefore have to install it from our own `brian-team` channel:

```
conda install -c brian-team brian2tools
```

```
python -m pip install brian2tools
```

1.1.6 Testing Brian

If you have the pytest testing utility installed, you can run Brian's test suite:

```python
import brian2
brian2.test()
```

It should end with "OK", showing a number of skipped tests but no errors or failures. For more control about the tests that are run see the *developer documentation on testing*.

1.2 Running Brian scripts

Brian scripts are standard Python scripts, and can therefore be run in the same way. For interactive, explorative work, you might want to run code in a jupyter notebook or in an ipython shell; for running finished code, you might want to execute scripts through the standard Python interpreter; finally, for working on big projects spanning multiple files, a dedicated integrated development environment for Python could be a good choice. We will briefly describe all these approaches and how they relate to Brian's examples and tutorial that are part of this documentation. Note that none of these approaches are specific to Brian, so you can also search for more information in any of the resources listed on the Python website.

- *Jupyter notebook*
- *IPython shell*
- *Python interpreter*
- *Integrated development environment (IDE)*

1.2.1 Jupyter notebook

> The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.
>
> (from jupyter.org)

Jupyter notebooks are a great tool to run Brian code interactively, and include the results of the simulations, as well as additional explanatory text in a common document. Such documents have the file ending `.ipynb`, and in Brian we use this format to store the *Tutorials*. These files can be displayed by github (see e.g. the first Brian tutorial), but in this case you can only see them as a static website, not edit or execute any of the code.

To make the full use of such notebooks, you have to run them using the jupyter infrastructure. The easiest option is to use the free mybinder.org web service, which allows you to try out Brian without installing it on your own machine. Links to run the tutorials on this infrastructure are provided as *"launch binder"* buttons on the *Tutorials* page, and also for each of the *Examples* at the top of the respective page (e.g. *Example: COBAHH*). To run notebooks on your own machine, you need an installation of the jupyter notebook software on your own machine, as well as Brian itself (see the *Installation* instructions for details). To open an existing notebook, you have to download it to your machine. For the Brian tutorials, you find the necessary links on the *Tutorials* page. When you have downloaded/installed everything necessary, you can start the jupyter notebook from the command line (using Terminal on OS X/Linux, Command Prompt on Windows):

```
jupyter notebook
```

this will open the "Notebook Dashboard" in your default browser, from which you can either open an existing notebook or create a new one. In the notebook, you can then execute individual "code cells" by pressing SHIFT+ENTER on your keyboard, or by pressing the play button in the toolbar.

For more information, see the jupyter notebook documentation.

1.2.2 IPython shell

An alternative to using the jupyter notebook is to use the interactive Python shell IPython, which runs in the Terminal/Command Prompt. You can use it to directly type Python code interactively (each line will be executed as soon as you press ENTER), or to run Python code stored in a file. Such files typically have the file ending .py. You can either create it yourself in a text editor of your choice (e.g. by copying&pasting code from one of the *Examples*), or by downloading such files from places such as github (e.g. the Brian examples), or ModelDB. You can then run them from within IPython via:

```
%run filename.py
```

1.2.3 Python interpreter

The most basic way to run Python code is to run it through the standard Python interpreter. While you can also use this interpreter interactively, it is much less convenient to use than the IPython shell or the jupyter notebook described above. However, if all you want to do is to run an existing Python script (e.g. one of the Brian *Examples*), then you can do this by calling:

```
python filename.py
```

in a Terminal/Command Prompt.

1.2.4 Integrated development environment (IDE)

Python is a widely used programming language, and is therefore support by a wide range of integrated development environments (IDE). Such IDEs provide features that are very convenient for developing complex projects, e.g. they integrate text editor and interactive Python console, graphical debugging tools, etc. Popular environments include Spyder, PyCharm, and Visual Studio Code, for an extensive list see the Python wiki.

1.3 Release notes

1.3.1 Brian 2.5.0.3

Another patch-level release that fixes incorrectly built Python wheels (the binary package used to install packages with pip). The wheels where mistakenly built against the most recent version of numpy (1.22), which made them incompatible with earlier versions of numpy. This release also fixes a few minor mistakes in the string representation of monitors, contributed by Felix Benjamin Kern.

1.3.2 Brian 2.5.0.2

A new patch-level release that fixes a missing #include in the synapse generation code for C++ standalone code. This does not matter for most compilers (in particular, it does not matter for the gcc, clang, and Visual Studio compilers that we use for testing on Linux, OS X, and Windows), but it can matter for projects like Brian2GeNN that build on top of Brian2 and use Nvidia's nvcc compiler. The release also fixes a minor string-formatting error (#1377), which led to quantities that were displayed without their units.

1.3.3 Brian 2.5.0.1

A new build to provide binary wheels for Python 3.10.

1.3.4 Brian 2.5

This new major release contains a large number of bug fixes and improvements, as well as important new features for synapse generation: the *Creating synapses with the generator syntax* can now create synapses "in both directions", and also supports random samples of fixed size. In addition, several contributors have helped to improve the documentation, in particular by adding several new *Examples*. We have also updated our test infrastructure and removed workarounds and warnings related to older, now unsupported, versions of Python. Our policy for supported Python and numpy versions now follows the NEP 29 policy adopted by most packages in the scientific Python ecosystem. This and other policies related to compatibility have been documented in *Compatibility and reproducibility*. As always, we recommend all users of Brian 2 to upgrade.

New features

- *Creating synapses with the generator syntax* has become more powerful: it is now possible to express pre-synaptic indices as a function of post-synaptic indices – previously, only the other direction was supported (#1294).

- Synapse generation can now make use of fixed-size random sampling (#1280). Together with the more powerful generator syntax, this finally makes it possible to have networks where each cell receives a fixed number of random inputs: `syn.connect(i='k for k in sample(N_pre, size=number_of_inputs)')`.

Selected improvements and bug fixes

- Fair default build flags on several architectures (#1277). Thanks to Étienne Mollier for contributing this feature.

- Better C++ compiler detection on UNIX systems, e.g. with Anaconda installations (#1304). Thanks to Jan Marker for this contribution.

- Fixed LaTeX output for newer sympy versions (#1299). Thanks to Sebastian Schmitt for reporting this issue. The problem and its fix is described in detail in this blog post.

- Fixed string representation for units (#1291). Recreating a unit from its string representation gave wrong results in some corner cases.

- Fix an error during the determination of appropriate C++ compiler flags on Windows with Python 3.9 (#1286), and fix the detection of a C99-compatible compiler on Windows (#1257). Thanks to Kyle Johnsen for reporting the errors and providing both fixes.

- More robust usage of external constants in C++ standalone code, avoiding clashes when the user defines constants with common names like `x` (#1279). Thanks to user `@wxie2013` for making us aware of this issue.

- Raise an error if summed variables refer to event-based variables (#1274) and a general rework of the dependency checks (#1328). Thanks to Rohith Varma Buddaraju for fixing this issue.

- Fix an error for deactivated spike-emitting objects (e.g. `NeuronGroup`, `PoissonGroup`). They continued to emit spikes despite `active=False` if they had spiked in the last time step of a previous run (#1319). Thanks to forum user Shencong for making us aware of the issue.

- Avoid warnings about deprecated numpy aliases (#1273).

- Avoid a warning about an "ignored attribute shape" in some interactive Python consoles (#1372).

- Check units for summed variables (#1361). Thanks to Jan-Hendrik Schleimer for reporting this issue.

- Do not raise an error if synapses use restore instead of Synapses.connect (#1359). Thanks to forum user SIbanez for reporting this issue.

- Fix indexing for sections in SpatialNeuron (#1358). Thanks to Sebastian Schmitt for reporting this issue

- Better error messages for missing threshold definition (#1363).

- Raise a useful error for `namespace` entries that start with an underscore instead of failing during compilation if the name clashes with built-in functions (#1362). Thanks to Denis Alevi for reporting this issue.

- Consistently use include/library directory preferences (#1353). The preferences can now be used to override the list of include/library directories, replacing the inconsistent behavior where they were either prepended (C++ standalone mode) or appended (Cython runtime mode) to the default list. Thanks to Denis Alevi for opening the discussion on this issue.

- Remove a warning about the difference between Python 2 and Python 3 semantics related to division (#1351).

- Do not generate spurious `-.o` files when checking compiler compatibility (#1348). For more details, see this blog post.

- Make `reset_to_defaults` work again, which was inadvertently broken in the Python 2 → 3 transition (#1342). Thanks to Denis Alevi for reporting and fixing this issue.

- The commands to run and compile the code in C++ standalone mode can now be changed via a preference (#1338). This can be useful to run/compile on clusters where jobs have to submitted with special commands. Thanks to Denis Alevi for contributing this feature.

Backward-incompatible changes

- The `default_preferences` file that was part of the Brian installation has been removed, since it could lead to problems when working with development versions of Brian, and was overwritten with each update (#1354). Users can still use a system-wide or per-directory preference file (see *Preferences*).

- The preferences *codegen.cpp.include_dirs*, *codegen.cpp.library_dirs*, and *codegen.cpp.runtime_library_dirs* now all replace the respective default values. Previously they where prepended (C++ standalone mode) or appended (Cython runtime mode). Users relying on a combination of the default values and their manually set values need to include the default value (e.g. `os.path.join(sys.prefix, 'include')`) manually.

Infrastructure and documentation improvements

- Tagging a release will now automatically upload the release to PyPI via a GitHub Action. Versions are automatically determined with versioneer (#1267) and include more detailed information when using a development version of Brian. See *Which version of Brian am I using?* for more details.

- The test suite has been moved to GitHub Actions for all operating systems (#1298). Thanks to Rohith Varma Buddaraju for working on this.

- New *Example: Jansen_Rit_1995_single_column* (#1347), contributed by Ruben Tikidji-Hamburyan.

- New *Example: spike_based_homeostasis* (#1331), contributed by Sebastian Schmitt.

- New *Example: COBAHH_approximated* (#1309), contributed by Sebastian Schmitt.

- Several new examples covering several Brian usage pattern, e.g. a *minimal C++ standalone script*, or demonstrations of running multiple simulations in parallel with *Cython* or *C++ standalone*, contributed by A. Ziaeemehr.

- Corrected units in *Example: Kremer_et_al_2011_barrel_cortex* (#1355). Thanks to Adam Willats for contributing this fix.

- Most of Brian's code base should now use a consistent string formatting style (#1364), documented in the *Coding conventions*.

- Test reports will now show the project directory path for C++ standalone projects (#1336). Thanks to Denis Alevi for contributing this feature.

- Fix the documentation for C++ compiler references (#1323, #1321). Thanks to Denis Alevi for fixing these issues.

- Examples are now listed in a deterministic order in the documentation (#1312), and their title is now correctly formatted in the restructured text source (#1311). Thanks to Felix C. Stegermann for contributing these fixes.

- Document how to plot model functions (e.g. time constants) in complex neuron models (#1308). Contributed by Sebastian Schmitt.

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Rohith Varma Buddaraju (@rohithvarma3000)

- Denis Alevi (@denisalevi)

- Dingkun.Liu (@DingkunLiu)

- Ruben Tikidji-Hamburyan (@rat-h)

- Sebastian Schmitt (@schmitts)

- @ramapati166

- Jan Marker (@jangmarker)

- Kyle Johnsen (@kjohnsen)

- Abolfazl Ziaeemehr (@Ziaeemehr)

- Felix Benjamin Kern (@kernfel)

- Yann Zerlaut (@yzerlaut)

- Adam (@Adam-Antios)

- @ShanqMa

- Ljubica Cimeša (@LjubicaCimesa)

- @adididi

- VigneswaranC (@Vigneswaran-Chandrasekaran)

- Nunna Lakshmi Saranya (@18sarru)

- Friedemann Zenke (@fzenke)

- @Alexis-Melot

- Adam Willats (@awillats)

- Felix C. Stegerman (@obfusk)

- Eugen Skrebenkov (@shcecter)

- Maurizio DE PITTA (@mdepitta)

- Simo ([@sivanni](#))
- Peter Quitta ([@peschn](#))
- Étienne Mollier ([@emollier](#))
- chaddy ([@chaddy1004](#))
- [@DePasquale99](#)
- [@albertalbesa](#)
- Christian Behrens ([@chbehrens](#))

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- forum user [Shencong](#)
- forum user [SIbanez](#)

1.3.5 Brian 2.4.1

This is a bugfix release with a number of small fixes and updates to the continuous integration testing.

Selected improvements and bug fixes

- The *`check_units()`* decorator can now express that some arguments need to have the same units. This mechanism is now used to check the units of the `clip()` function ([#1234](#)). Thanks to Felix Kern for notifying us of this issue.
- Using *`SpatialNeuron`* with Cython no longer raises an unnecessary warning when the `scipy` library is not installed ([#1230](#)).
- Raise an error for references to `N_incoming` or `N_outgoing` in calls to *`Synapses.connect`*. This use is ill-defined and led to compilation errors in previous versions ([#1227](#)). Thanks to Denis Alevi for making us aware of this issue.

Infrastructure and documentation improvements

- Brian no longer officially supports installation on 32bit operating systems. Installation via `pip` will probably still work, but we are no longer testing this configuration ([#1232](#)).
- Automatic continuous integration tests for Windows now use the [Microsoft Azure Pipeline](#) infrastructure instead of [Appveyor](#). This should speed up tests by running different configurations in parallel ([#1233](#)).
- Fix an issue in the test suite that did not handle `NotImplementedError` correctly anymore after the changes introduced with [#1196](#).

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Denis Alevi (@denisalevi)

- SK (@akatav)

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Felix B. Kern

1.3.6 Brian 2.4

This new release contains a large number of small improvements and bug fixes. We recommend all users of Brian 2 to upgrade. The biggest code change of this new version is that Brian is now Python-3 only (thanks to Ben Evans for working on this).

Selected improvements and bug fixes

- Removing objects from networks no longer fails (#1151). Thanks to Wilhelm Braun for reporting the issue.

- Point currents marked as `constant over dt` are now correctly handled (#1160). Thanks to Andrew Brughera for reporting the issue.

- Elapsed and estimated remaining time are now formatted as hours/minutes/etc. in standalone mode as well (#1162). Thanks to Rahul Kumar Gupta, Syed Osama Hussain, Bhuwan Chandra, and Vigneswaran Chandrasekaran for working on this issue as part of the GSoC 2020 application process.

- To prevent log files filling up the disk (#1188), their file size is now limited to 10MB (configurable via the *logging.file_log_max_size* preference). Thanks to Rike-Benjamin Schuppner for contributing this feature.

- Add more complete support for operations on *VariableView* attributes. Previously, operations like `group.v**2` failed and required the workaround `group.v[:]**2` (#1195)

- Fix a number of compatibility issues with newer versions of numpy and sympy, and document our policy on *Compatibility and reproducibility*.

- File locking (used to avoid problems when running multiple simulations in parallel) is now based on Benedikt Schmitt's py-filelock package, which should hopefully make it more robust.

- String expressions in *Synapses.connect* are now checked for syntactic correctness before handing them over to the code generation process, improving error messages. Thanks to Denis Alevi for making us aware of this issue. (#1224)

- Avoid duplicate messages in "chained" exceptions. Also introduces a new preference *logging.display_brian_error_message* to switch off the "Brian 2 encountered an unexpected error" message (#1196).

- Brian's unit system now correctly deals with matrix multiplication, including the `@` operator (#1216). Thanks to @kjohnsen for reporting this issue.

- Avoid turning all integer numbers in equations into floating point values (#1202). Thanks to Marco K. for making us aware of this issue.

- New attributes *Synapses.N_outgoing_pre* and *Synapses.N_incoming_post* to access the number of synapses per pre-/post-synaptic cell (see *Accessing synaptic variables* for details; #1225)

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg ([@mstimberg](#))

- Ben Evans ([@bdevans](#))

- Dan Goodman ([@thesamovar](#))

- Denis Alevi ([@denisalevi](#))

- Rike-Benjamin Schuppner ([@Debilski](#))

- Syed Osama Hussain ([@Syed-Osama-Hussain](#))

- VigneswaranC ([@Vigneswaran-Chandrasekaran](#))

- Tushar ([@smalltimer](#))

- Felix Hoffmann ([@felix11h](#))

- Rahul Kumar Gupta ([@rahuliitg](#))

- Dominik Spicher ([@dspicher](#))

- [@nfzd](#)

- [@Snow-Crash](#)

- [@cnjackhu](#)

- [@neurologic](#)

- [@kjohnsen](#)

- Ashwin Viswanathan Kannan ([@ashwin4ever](#))

- Bhuwan Chandra ([@zeph1yr](#))

- Wilhelm Braun ([@wilhelmbraun](#))

- [@cortical-iv](#)

- Eugen Skrebenkov ([@shcecter](#))

- [@Aman-A](#)

- Felix Benjamin Kern ([@kernfel](#))

- Francesco Battaglia ([@fpbattaglia](#))

- Shivam Chitnis ([@shivChitinous](#))

- Marco K. ([@spokli](#))

- [@jcmharry](#)

- Friedemann Zenke ([@fzenke](#))

- [@Adam-Antios](#)

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Andrew Brughera

- William Xavier

1.3.7 Brian 2.3

This release contains the usual mix of bug fixes and new features (see below), but also makes some important changes to the Brian 2 code base to pave the way for the full Python 2 -> 3 transition (the source code is now directly compatible with Python 2 and Python 3, without the need for any translation at install time). Please note that this release will be the last release that supports Python 2, given that Python 2 reaches end-of-life in January 2020. Brian now also uses pytest as its testing framework, since the previously used `nose` package is not maintained anymore. Since brian2hears has been released as an independent package, using `brian2.hears` as a "bridge" to Brian 1's `brian.hears` package is now deprecated.

Finally, the Brian project has adopted the "Contributor Covenant" *Contributor Covenant Code of Conduct*, pledging "to make participation in our community a harassment-free experience for everyone".

New features

- The `restore()` function can now also restore the state of the random number generator, allowing for exact reproducibility of stochastic simulations (#1134)

- The functions `expm1()`, `log1p()`, and `exprel()` can now be used (#1133)

- The system for calling random number generating functions has been generalized (see *Functions with context-dependent return values*), and a new `poisson` function for Poisson-distrubted random numbers has been added (#1111)

- New versions of Visual Studio are now supported for standalone mode on Windows (#1135)

Selected improvements and bug fixes

- `run_regularly` operations are now included in the network, even if they are created after the parent object was added to the network (#1009). Contributed by Vigneswaran Chandrasekaran.

- No longer incorrectly classify some equations as having "multiplicative noise" (#968). Contributed by Vigneswaran Chandrasekaran.

- Brian is now compatible with Python 3.8 (#1130), and doctests are compatible with numpy 1.17 (#1120)

- Progress reports for repeated runs have been fixed (#1116), thanks to Ronaldo Nunes for reporting the issue.

- `SpikeGeneratorGroup` now correctly works with `restore()` (#1084), thanks to Tom Achache for reporting the issue.

- An indexing problem in `PopulationRateMonitor` has been fixed (#1119).

- Handling of equations referring to `-inf` has been fixed (#1061).

- Long simulations recording more than ~2 billion data points no longer crash with a segmentation fault (#1136), thanks to Rike-Benjamin Schuppner for reporting the issue.

Backward-incompatible changes

- The fix for `run_regularly` operations (#1009, see above) entails a change in how objects are stored within `Network` objects. Previously, `Network.objects` stored a complete list of all objects, including objects such as `StateUpdater` that – often invisible to the user – are a part of major objects such as `NeuronGroup`. Now, `Network.objects` only stores the objects directly provided by the user (`NeuronGroup`, `Synapses`, `StateMonitor`, …), the dependent objects (`StateUpdater`, `Thresholder`, …) are taken into account at the time of the run. This might break code in some corner cases, e.g. when removing a `StateUpdater` from `Network.objects` via `Network.remove`.

- The `brian2.hears` interface to Brian 1's `brian.hears` package has been deprecated.

Infrastructure and documentation improvements

- The same code base is used on Python 2 and Python 3 (#1073).

- The test framework uses `pytest` (#1127).

- We have adapoted a Code of Conduct (#1113), thanks to Tapasweni Pathak for the suggestion.

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Vigneswaran Chandrasekaran (@Vigneswaran-Chandrasekaran)

- Moritz Orth (@morth)

- Tristan Stöber (@tristanstoeber)

- @ulyssek

- Wilhelm Braun (@wilhelmbraun)

- @flomlo

- Rike-Benjamin Schuppner (@Debilski)

- @sdeiss

- Ben Evans (@bdevans)

- Tapasweni Pathak (@tapaswenipathak)

- @jonathanoesterle

- Richard C Gerkin (@rgerkin)

- Christian Behrens (@chbehrens)

- Romain Brette (@romainbrette)

- XiaoquinNUDT (@XiaoquinNUDT)

- Dylan Muir (@DylanMuir)

- Aleksandra Teska (@alTeska)

- Felix Z. Hoffmann (@felix11h)

- @baixiaotian63648995

- Carlos de la Torre (@c-torre)

- Sam Mathias (@sammosummo)

- @Marghepano

- Simon Brodeur (@sbrodeur)

- Alex Dimitrov (@adimitr)

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Ronaldo Nunes

- Tom Achache

1.3.8 Brian 2.2.2.1

This is a bug-fix release that fixes several bugs and adds a few minor new features. We recommend all users of Brian 2 to upgrade.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

[Note that the original upload of this release was version 2.2.2, but due to a mistake in the released archive, it has been uploaded again as version 2.2.2.1]

Selected improvements and bug fixes

- Fix an issue with the synapses generator syntax (#1037).

- Fix an incorrect error when using a `SpikeGeneratorGroup` with a long period (#1041). Thanks to Kévin Cuallado-Keltsch for reporting this issue.

- Improve the performance of `SpikeGeneratorGroup` by avoiding a conversion from time to integer time step (#1043). This time step is now also available to user code as `t_in_timesteps`.

- Function definitions for weave/Cython/C++ standalone can now declare additional header files and libraries. They also support a new `sources` argument to use a function definition from an external file. See the *Functions* documentation for details.

- For convenience, single-neuron subgroups can now be created with a single index instead of with a slice (e.g. `neurongroup[3]` instead of `neurongroup[3:4]`).

- Fix an issue when `-inf` is used in an equation (#1061).

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Felix Z. Hoffmann (@Felix11H)

- @wjx0914

- Kévin Cuallado-Keltsch (@kevincuallado)

- Romain Cazé (@rcaze)

- Daphne (@daphn3cor)

- Erik (@parenthetical-e)

- @RahulMaram

- Eghbal Hosseini (@eghbalhosseini)

- Martino Sorbaro (@martinosorb)

- Mihir Vaidya (@MihirVaidya94)

- @hellolingling

- Volodimir Slobodyanyuk (@vslobody)

- Peter Duggins (@psipeter)

1.3.9 Brian 2.2.1

This is a bug-fix release that fixes a few minor bugs and incompatibilites with recent versions of the dependencies. We recommend all users of Brian 2 to upgrade.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Selected improvements and bug fixes

- Work around problems with the latest version of `py-cpuinfo` on Windows (#990, #1020) and no longer require it for Linux and OS X.

- Avoid warnings with newer versions of Cython (#1030) and correctly build the Cython spike queue for Python 3.7 (#1026), thanks to Fleur Zeldenrust and Ankur Sinha for reporting these issues.

- Fix error messages for `SyntaxError` exceptions in jupyter notebooks (##964).

Dependency and packaging changes

- Conda packages in conda-forge are now avaible for Python 3.7 (but no longer for Python 3.5).

- Linux and OS X no longer depend on the `py-cpuinfo` package.

- Source packages on pypi now require a recent Cython version for installation.

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Christopher (@Chris-Currin)

- Peter Duggins (@psipeter)

- Paola Suárez (@psrmx)

- Ankur Sinha (@sanjayankur31)

- @JingjinW

- Denis Alevi (@denisalevi)

- @lemonade117

- @wjx0914

- Sven Leach (@SvennoNito)

- svadams (@svadams)

- @ghaessig

- Varshith Sreeramdass (@varshiths)

1.3.10 Brian 2.2

This releases fixes a number of important bugs and comes with a number of performance improvements. It also makes sure that simulation no longer give platform-dependent results for certain corner cases that involve the division of integers. These changes can break backwards-compatiblity in certain cases, see below. We recommend all users of Brian 2 to upgrade.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Selected improvements and bug fixes

- Divisions involving integers now use floating point division, independent of Python version and code generation target. The `//` operator can now used in equations and expressions to denote flooring division (#984).

- Simulations can now use single precision instead of double precision floats in simulations (#981, #1004). This is mostly intended for use with GPU code generation targets.

- The *`timestep`*, introduced in version 2.1.3, was further optimized for performance, making the refractoriness calculation faster (#996).

- The `lastupdate` variable is only automatically added to synaptic models when event-driven equations are used, reducing the memory and performance footprint of simple synaptic models (#1003). Thanks to Denis Alevi for bringing this up.

- A `from brian2 import *` imported names unrelated to Brian, and overwrote some Python builtins such as `dir` (#969). Now, fewer names are imported (but note that this still includes numpy and plotting tools: *Importing Brian*).

- The `exponential_euler` state updater is no longer failing for systems of equations with differential equations that have trivial, constant right-hand-sides (#1010). Thanks to Peter Duggins for making us aware of this issue.

Backward-incompatible changes

- Code that divided integers (e.g. `N/10`) with a C-based code generation target, or with the `numpy` target on Python 2, will now use floating point division instead of flooring division (i.e., Python 3 semantics). A warning will notify the user of this change, use either the flooring division operator (`N//10`), or the `int` function (`int(N/10)`) to make the expression unambiguous.

- Code that directly referred to the `lastupdate` variable in synaptic statements, without using any event-driven variables, now has to manually add `lastupdate :  second` to the equations and update the variable at the end of `on_pre` and/or `on_post` with `lastupdate = t`.

- Code that relied on `from brian2 import *` also importing unrelated names such as `sympy`, now has to import such names explicitly.

Documentation improvements

- Various small fixes and additions (e.g. installation instructions, available functions, fixes in examples)

- A new example, *Izhikevich 2007*, provided by Guillaume Dumas.

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Denis Alevi (@denisalevi)

- Thomas Nowotny (@tnowotny)

- @neworderofjamie

- Paul Brodersen (@paulbrodersen)

- @matrec4

- svadams (@svadams)

- XiaoquinNUDT (@XiaoquinNUDT)

- Peter Duggins (@psipeter)

- @nh17937

- Patrick Nave (@pnave95)

- @AI-pha

- Guillaume Dumas (@deep-introspection)

- @godelicbach

- @galharth

1.3.11 Brian 2.1.3.1

This is a bug-fix release that fixes two bugs in the recent 2.1.3 release:

- Fix an inefficiency in the newly introduced `timestep` function when using the `numpy` target (#965)

- Fix inefficiencies in the unit system that could lead to slow operations and high memory use (#967). Thanks to Kaustab Pal for making us aware of the issue.

1.3.12 Brian 2.1.3

This is a bug-fix release that fixes a number of important bugs (see below), but does not introduce any new features. We recommend all users of Brian 2 to upgrade.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Selected improvements and bug fixes

- The Cython cache on disk now uses significantly less space by deleting unnecessary source files (set the *codegen.runtime.cython.delete_source_files* preference to `False` if you want to keep these files for debugging). In addition, a warning will be given when the Cython or weave cache exceeds a configurable size (*codegen.max_cache_dir_size*). The `clear_cache` function is provided to delete files from the cache (#914).

- The C++ standalone mode now respects the `profile` option and therefore no longer collects profiling information by default. This can speed up simulations in certain cases (#935).

- The exact number of time steps that a neuron stays in the state of refractoriness after a spike could vary by up to one time step when the requested refractory time was a multiple of the simulation time step. With this fix, the number of time steps is ensured to be as expected by making use of a new `timestep` function that avoids floating point rounding issues (#949, first reported by @zhouyanasd in issue #943).

- When `restore()` was called twice for a network, spikes that were not yet delivered to their target were not restored correctly (#938, reported by @zhouyanasd).

- `SpikeGeneratorGroup` now uses a more efficient method for sorting spike indices and times, leading to a much faster preparation time for groups that store many spikes (#948).

- Fix a memory leak in `TimedArray` (#923, reported by Wilhelm Braun).

- Fix an issue with summed variables targetting subgroups (#925, reported by @AI-pha).

- Fix the use of `run_regularly` on subgroups (#922, reported by @AI-pha).

- Improve performance for `SpatialNeuron` by removing redundant computations (#910, thanks to Moritz Augustin for making us aware of the issue).

- Fix linked variables that link to scalar variables (#916)

- Fix warnings for numpy 1.14 and avoid compilation issues when switching between versions of numpy (#913)

- Fix problems when using logical operators in code generated for the numpy target which could lead to issues such as wrongly connected synapses (#901, #900).

Backward-incompatible changes

- No longer allow `delay` as a variable name in a synaptic model to avoid ambiguity with respect to the synaptic delay. Also no longer allow access to the `delay` variable in synaptic code since there is no way to distinguish between pre- and post-synaptic delay (#927, reported by Denis Alevi).

- Due to the changed handling of refractoriness (see bug fixes above), simulations that make use of refractoriness will possibly no longer give exactly the same results. The preference *legacy.refractory_timing* can be set to `True` to reinstate the previous behaviour.

Infrastructure and documentation improvements

- From this version on, conda packages will be available on conda-forge. For a limited time, we will copy over packages to the `brian-team` channel as well.

- Conda packages are no longer tied to a specific numpy version (PR #954)

- New example (*Brunel & Wang, 2001*) contributed by Teo Stocco and Alex Seeholzer.

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Teo Stocco (@zifeo)

- Dylan Muir (@DylanMuir)

- scarecrow (@zhouyanasd)

- @fuadfukhasyi

- Aditya Addepalli (@Dyex719)

- Kapil kumar (@kapilkd13)

- svadams (@svadams)

- Vafa Andalibi (@Vafa-Andalibi)

- Sven Leach (@SvennoNito)

- @matrec4

- @jarishna

- @AI-pha

- @xdzhangxuejun

- Denis Alevi (@denisalevi)

- Paul Pfeiffer (@pfeffer90)

- Romain Brette (@romainbrette)

- @hustyanghui

- Adrien F. Vincent (@afvincent)

- @ckemere

- @evearmstrong

- Paweł Kopeć (@pawelkopec)

- Moritz Augustin (@moritzaugustin)

- Bart (@louwers)

- @amarsdd

- @ttxtea

- Maria Cervera (@MariaCervera)

- ouyangxinrong ([@longzhixin](#))

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Wilhelm Braun

1.3.13 Brian 2.1.2

This is another bug fix release that fixes a major bug in `Equations`' substitution mechanism ([#896](#)). Thanks to Teo Stocco for reporting this issue.

1.3.14 Brian 2.1.1

This is a bug fix release that re-activates parts of the caching mechanism for code generation that had been erroneously deactivated in the previous release.

1.3.15 Brian 2.1

This release introduces two main new features: a new "GSL integration" mode for differential equation that offers to integrate equations with variable-timestep methods provided by the GNU Scientific Library, and caching for the run preparation phase that can significantly speed up simulations. It also comes with a newly written tutorial, as well as additional documentation and examples.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

New features

- New numerical integration methods with variable time-step integration, based on the GNU Scientific Library (see *Numerical integration*). Contributed by [Charlee Fletterman](#), supported by 2017's [Google Summer of Code](#) program.

- New caching mechanism for the code generation stage (application of numerical integration algorithms, analysis of equations and statements, etc.), reducing the preparation time before the actual run, in particular for simulations with multiple `run()` statements.

Selected improvements and bug fixes

- Fix a rare problem in Cython code generation caused by missing type information ([#893](#))

- Fix warnings about improperly closed files on Python 3.6 ([#892](#); reported and fixed by [Teo Stocco](#))

- Fix an error when using numpy integer types for synaptic indexing ([#888](#))

- Fix an error in numpy codegen target, triggered when assigning to a variable with an unfulfilled condition ([#887](#))

- Fix an error when repeatedly referring to subexpressions in multiline statements ([#880](#))

- Shorten long arrays in warning messages ([#874](#))

- Enable the use of `if` in the shorthand generator syntax for `Synapses.connect` ([#873](#))

- Fix the meaning of `i` and `j` in synapses connecting to/from other synapses ([#854](#))

Backward-incompatible changes and deprecations

- In C++ standalone mode, information about the number of synapses and spikes will now only be displayed when built with `debug=True` (#882).

- The `linear` state updater has been renamed to `exact` to avoid confusion (#877). Users are encouraged to use `exact`, but the name `linear` is still available and does not raise any warning or error for now.

- The `independent` state updater has been marked as deprecated and might be removed in future versions.

Infrastructure and documentation improvements

- A new, more advanced, *tutorial* "about managing the slightly more complicated tasks that crop up in research problems, rather than the toy examples we've been looking at so far."

- Additional documentation on *Custom events* and *Converting from integrated form to ODEs* (including example code for typical synapse models).

- New example code reproducing published findings (*Platkiewicz and Brette, 2011*; *Stimberg et al., 2018*)

- Fixes to the sphinx documentation creation process, the documentation can be downloaded as a PDF once again (705 pages!)

- Conda packages now have support for numpy 1.13 (but support for numpy 1.10 and 1.11 has been removed)

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)

- Charlee Fletterman (@CharleeSF)

- Dan Goodman (@thesamovar)

- Teo Stocco (@zifeo)

- @k47h4

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Chaofei Hong

- Lucas ("lucascdst")

1.3.16 Brian 2.0.2.1

Fixes a bug in the tutorials' HMTL rendering on readthedocs.org (code blocks were not displayed). Thanks to Flora Bouchacourt for making us aware of this problem.

1.3.17 Brian 2.0.2

New features

- `molar` and `liter` (as well as `litre`, scaled versions of the former, and a few useful abbreviations such as `mM`) have been added as new units (#574).

- A new module *brian2.units.constants* provides physical constants such as the Faraday constants or the gas constant (see *Constants* for details).

- *SpatialNeuron* now supports non-linear membrane currents (e.g. Goldman–Hodgkin–Katz equations) by linearizing them with respect to v.

- Multi-compartmental models can access the capacitive current via `Ic` in their equations (#677)

- A new function *scheduling_summary()* that displays information about the scheduling of all objects (see *Scheduling* for details).

- Introduce a new preference to pass arguments to the `make`/`nmake` command in C++ standalone mode (*devices.cpp_standalone.extra_make_args_unix* for Linux/OS X and *devices.cpp_standalone.extra_make_args_windows* for Windows). For Linux/OS X, this enables parallel compilation by default.

- Anaconda packages for Brian 2 are now available for Python 3.6 (but Python 3.4 support has been removed).

Selected improvements and bug fixes

- Work around low performance for certain C++ standalone simulations on Linux, due to a bug in glibc (see #803). Thanks to Oleg Strikov (@xj8z) for debugging this issue and providing the workaround that is now in use.

- Make exact integration of `event-driven` synaptic variables use the `linear` numerical integration algorithm (instead of `independent`), fixing rare occasions where integration failed despite the equations being linear (#801).

- Better error messages for incorrect unit definitions in equations.

- Various fixes for the internal representation of physical units and the unit registration system.

- Fix a bug in the assignment of state variables in subtrees of *SpatialNeuron* (#822)

- Numpy target: fix an indexing error for a *SpikeMonitor* that records from a subgroup (#824)

- Summed variables targeting the same post-synaptic variable now raise an error (previously, only the one executed last was taken into account, see #766).

- Fix bugs in synapse generation affecting Cython (#781) respectively numpy (#835)

- C++ standalone simulations with many objects no longer fail on Windows (#787)

Backwards-incompatible changes

- `celsius` has been removed as a unit, because it was ambiguous in its relation to `kelvin` and gave wrong results when used as an absolute temperature (and not a temperature difference). For temperature differences, you can directly replace `celsius` by `kelvin`. To convert an absolute temperature in degree Celsius to Kelvin, add the `zero_celsius` constant from *brian2.units.constants* (#817).

- State variables are no longer allowed to have names ending in `_pre` or `_post` to avoid confusion with references to pre- and post-synaptic variables in *Synapses* (#818)

Changes to default settings

- In C++ standalone mode, the `clean` argument now defaults to `False`, meaning that `make clean` will not be executed by default before building the simulation. This avoids recompiling all files for unchanged simulations that are executed repeatedly. To return to the previous behaviour, specify `clean=True` in the `device.build` call (or in `set_device` if your script does not have an explicit `device.build`).

Contributions

Github code, documentation, and issue contributions (ordered by the number of contributions):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Thomas McColgan (@phreeza)
- Daan Sprenkels (@dsprenkels)
- Romain Brette (@romainbrette)
- Oleg Strikov (@xj8z)
- Charlee Fletterman (@CharleeSF)
- Meng Dong (@whenov)
- Denis Alevi (@denisalevi)
- Mihir Vaidya (@MihirVaidya94)
- Adam (@ffa)
- Sourav Singh (@souravsingh)
- Nick Hale (@nik849)
- Cody Greer (@Cody-G)
- Jean-Sébastien Dessureault (@jsdessureault)
- Michele Giugliano (@mgiugliano)
- Teo Stocco (@zifeo)
- Edward Betts (@EdwardBetts)

Other contributions outside of github (ordered alphabetically, apologies to anyone we forgot…):

- Christopher Nolan
- Regimantas Jurkus
- Shailesh Appukuttan

1.3.18 Brian 2.0.1

This is a bug-fix release that fixes a number of important bugs (see below), but does not introduce any new features. We recommend all users of Brian 2 to upgrade.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Improvements and bug fixes

- Fix *PopulationRateMonitor* for recordings from subgroups (#772)

- Fix *SpikeMonitor* for recordings from subgroups (#777)

- Check that string expressions provided as the `rates` argument for *PoissonGroup* have correct units.

- Fix compilation errors when multiple run statements with different `report` arguments are used in C++ standalone mode.

- Several documentation updates and fixes

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Alex Seeholzer (@flinz)

- Meng Dong (@whenov)

Testing, suggestions and bug reports (ordered alphabetically, apologies to anyone we forgot…):

- Myung Seok Shim

- Pamela Hathway

1.3.19 Brian 2.0 (changes since 1.4)

Major new features

- Much more flexible model definitions. The behaviour of all model elements can now be defined by arbitrary equations specified in standard mathematical notation.

- Code generation as standard. Behind the scenes, Brian automatically generates and compiles C++ code to simulate your model, making it much faster.

- "Standalone mode". In this mode, Brian generates a complete C++ project tree that implements your model. This can be then be compiled and run entirely independently of Brian. This leads to both highly efficient code, as well as making it much easier to run simulations on non-standard computational hardware, for example on robotics platforms.

- Multicompartmental modelling.

- Python 2 and 3 support.

New features

- Installation should now be much easier, especially if using the Anaconda Python distribution. See *Installation*.

- Many improvements to *Synapses* which replaces the old Connection object in Brian 1. This includes: synapses that are triggered by non-spike events; synapses that target other synapses; huge speed improvements thanks to using code generation; new "generator syntax" when creating synapses is much more flexible and efficient. See *Synapses*.

- New model definitions allow for much more flexible refractoriness. See *Refractoriness*.

- *SpikeMonitor* and *StateMonitor* are now much more flexible, and cover a lot of what used to be covered by things like MultiStateMonitor, etc. See *Recording during a simulation*.

- Multiple event types. In addition to the default spike event, you can create arbitrary events, and have these trigger code blocks (like reset) or synaptic events. See *Custom events*.

- New units system allows arrays to have units. This eliminates the need for a lot of the special casing that was required in Brian 1. See *Physical units*.

- Indexing variable by condition, e.g. you might write G.v['x>0'] to return all values of variable v in *Neuron- Group* G where the group's variable x>0. See *State variables*.

- Correct numerical integration of stochastic differential equations. See *Numerical integration*.

- "Magic" *run()* system has been greatly simplified and is now much more transparent. In addition, if there is any ambiguity about what the user wants to run, an erorr will be raised rather than making a guess. This makes it much safer. In addition, there is now a *store()*/*restore()* mechanism that simplifies restarting simulations and managing separate training/testing runs. See *Running a simulation*.

- Changing an external variable between runs now works as expected, i.e. something like tau=1*ms; run(100*ms); tau=5*ms; run(100*ms). In Brian 1 this would have used tau=1*ms for both runs. More generally, in Brian 2 there is now better control over namespaces. See *Namespaces*.

- New "shared" variables with a single value shared between all neurons. See *Shared variables*.

- New *Group.run_regularly* method for a codegen-compatible way of doing things that used to be done with *network_operation()* (which can still be used). See *Regular operations*.

- New system for handling externally defined functions. They have to specify which units they accept in their arguments, and what they return. In addition, you can easily specify the implementation of user-defined functions in different languages for code generation. See *Functions*.

- State variables can now be defined as integer or boolean values. See *Equations*.

- State variables can now be exported directly to Pandas data frame. See *Storing state variables*.

- New generalised "flags" system for giving additional information when defining models. See *Flags*.

- *TimedArray* now allows for 2D arrays with arbitrary indexing. See *Timed arrays*.

- Better support for using Brian in IPython/Jupyter. See, for example, *start_scope()*.

- New preferences system. See *Preferences*.

- Random number generation can now be made reliably reproducible. See *Random numbers*.

- New profiling option to see which parts of your simulation are taking the longest to run. See *Profiling*.

- New logging system allows for more precise control. See *Logging*.

- New ways of importing Brian for advanced Python users. See *Importing Brian*.

- Improved control over the order in which objects are updated during a run. See *Custom progress reporting*.

- Users can now easily define their own numerical integration methods. See *State update*.

- Support for parallel processing using the OpenMP version of standalone mode. Note that all Brian tests pass with this, but it is still considered to be experimental. See *Multi-threading with OpenMP*.

Backwards incompatible changes

See *Detailed Brian 1 to Brian 2 conversion notes*.

Behind the scenes changes

- All user models are now passed through the code generation system. This allows us to be much more flexible about introducing new target languages for generated code to make use of non-standard computational hardware. See *Code generation*.

- New standalone/device mode allows generation of a complete project tree that can be compiled and built independently of Brian and Python. This allows for even more flexible use of Brian on non-standard hardware. See *Devices*.

- All objects now have a unique name, used in code generation. This can also be used to access the object through the `Network` object.

Contributions

Full list of all Brian 2 contributors, ordered by the time of their first contribution:

- Dan Goodman (@thesamovar)
- Marcel Stimberg (@mstimberg)
- Romain Brette (@romainbrette)
- Cyrille Rossant (@rossant)
- Victor Benichoux (@victorbenichoux)
- Pierre Yger (@yger)
- Werner Beroux (@wernight)
- Konrad Wartke (@Kwartke)
- Daniel Bliss (@dabliss)
- Jan-Hendrik Schleimer (@ttxtea)
- Moritz Augustin (@moritzaugustin)
- Romain Cazé (@rcaze)
- Dominik Krzemiński (@dokato)
- Martino Sorbaro (@martinosorb)
- Benjamin Evans (@bdevans)

1.3.20 Brian 2.0 (changes since 2.0rc3)

New features

- A new flag `constant over dt` can be applied to subexpressions to have them only evaluated once per timestep (see *Models and neuron groups*). This flag is mandatory for stateful subexpressions, e.g. expressions using `rand()` or `randn()`. (#720, #721)

Improvements and bug fixes

- Fix *EventMonitor.values* and *SpikeMonitor.spike_trains* to always return sorted spike/event times (#725).

- Respect the `active` attribute in C++ standalone mode (#718).

- More consistent check of compatible time and dt values (#730).

- Attempting to set a synaptic variable or to start a simulation with synapses without any preceding connect call now raises an error (#737).

- Improve the performance of coordinate calculation for *Morphology* objects, which previously made plotting very slow for complex morphologies (#741).

- Fix a bug in *SpatialNeuron* where it did not detect non-linear dependencies on v, introduced via point currents (#743).

Infrastructure and documentation improvements

- An interactive demo, tutorials, and examples can now be run in an interactive jupyter notebook on the mybinder platform, without any need for a local Brian installation (#736). Thanks to Ben Evans for the idea and help with the implementation.

- A new extensive guide for converting Brian 1 simulations to Brian 2 user coming from Brian 1: *Changes for Brian 1 users*

- A re-organized *User's guide*, with clearer indications which information is important for new Brian users.

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Benjamin Evans (@bdevans)

Testing, suggestions and bug reports (ordered alphabetically, apologies to anyone we forgot…):

- Chaofei Hong

- Daniel Bliss

- Jacopo Bono

- Ruben Tikidji-Hamburyan

1.3.21 Brian 2.0rc3

This is another "release candidate" for Brian 2.0 that fixes a range of bugs and introduces better support for random numbers (see below). We are getting close to the final Brian 2.0 release, the remaining work will focus on bug fixes, and better error messages and documentation.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

New features

- Brian now comes with its own `seed()` function, allowing to seed the random number generator and thereby to make simulations reproducible. This function works for all code generation targets and in runtime and standalone mode. See *Random numbers* for details.

- Brian can now export/import state variables of a group or a full network to/from a pandas `DataFrame` and comes with a mechanism to extend this to other formats. Thanks to Dominik Krzemiński for this contribution (see #306).

Improvements and bug fixes

- Use a Mersenne-Twister pseudorandom number generator in C++ standalone mode, replacing the previously used low-quality random number generator from the C standard library (see #222, #671 and #706).

- Fix a memory leak in code running with the weave code generation target, and a smaller memory leak related to units stored repetitively in the `UnitRegistry`.

- Fix a difference of one timestep in the number of simulated timesteps between runtime and standalone that could arise for very specific values of dt and t (see #695).

- Fix standalone compilation failures with the most recent gcc version which defaults to C++14 mode (see #701)

- Fix incorrect summation in synapses when using the `(summed)` flag and writing to *pre*-synaptic variables (see #704)

- Make synaptic pathways work when connecting groups that define nested subexpressions, instead of failing with a cryptic error message (see #707).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

- Dominik Krzemiński (@dokato)

- Dan Goodman (@thesamovar)

- Martino Sorbaro (@martinosorb)

Testing, suggestions and bug reports (ordered alphabetically, apologies to anyone we forgot…):

- Craig Henriquez

- Daniel Bliss

- David Higgins

- Gordon Erlebacher

- Max Gillett

- Moritz Augustin
- Sami Abdul-Wahid

1.3.22 Brian 2.0rc1

This is a bug fix release that we release only about two weeks after the previous release because that release introduced a bug that could lead to wrong integration of stochastic differential equations. Note that standard neuronal noise models were not affected by this bug, it only concerned differential equations implementing a "random walk". The release also fixes a few other issues reported by users, see below for more information.

Improvements and bug fixes

- Fix a regression from 2.0b4: stochastic differential equations without any non-stochastic part (e.g. `dx/dt = xi/sqrt(ms)`) were not integrated correctly (see #686).
- Repeatedly calling `restore()` (or `Network.restore`) no longer raises an error (see #681).
- Fix an issue that made `PoissonInput` refuse to run after a change of dt (see #684).
- If the `rates` argument of `PoissonGroup` is a string, it will now be evaluated at every time step instead of once at construction time. This makes time-dependent rate expressions work as expected (see #660).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

Testing, suggestions and bug reports (ordered alphabetically, apologies to anyone we forgot…):

- Cian O'Donnell
- Daniel Bliss
- Ibrahim Ozturk
- Olivia Gozel

1.3.23 Brian 2.0rc

This is a release candidate for the final Brian 2.0 release, meaning that from now on we will focus on bug fixes and documentation, without introducing new major features or changing the syntax for the user. This release candidate itself *does* however change a few important syntax elements, see "Backwards-incompatible changes" below.

As always, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- New "generator syntax" to efficiently generate synapses (e.g. one-to-one connections), see *Creating synapses* for more details.

- For synaptic connections with multiple synapses between a pair of neurons, the number of the synapse can now be stored in a variable, allowing its use in expressions and statements (see *Creating synapses*).

- `Synapses` can now target other `Synapses` objects, useful for some models of synaptic modulation.

- The `Morphology` object has been completely re-worked and several issues have been fixed. The new `Section` object allows to model a section as a series of truncated cones (see *Creating a neuron morphology*).

- Scripts with a single `run()` call, no longer need an explicit `device.build()` call to run with the C++ standalone device. A `set_device()` in the beginning is enough and will trigger the `build` call after the run (see *Standalone code generation*).

- All state variables within a `Network` can now be accessed by `Network.get_states` and `Network.set_states` and the `store()`/`restore()` mechanism can now store the full state of a simulation to disk.

- Stochastic differential equations with multiplicative noise can now be integrated using the Euler-Heun method (`heun`). Thanks to Jan-Hendrik Schleimer for this contribution.

- Error messages have been significantly improved: errors for unit mismatches are now much clearer and error messages triggered during the intialization phase point back to the line of code where the relevant object (e.g. a `NeuronGroup`) was created.

- `PopulationRateMonitor` now provides a `smooth_rate` method for a filtered version of the stored rates.

Improvements and bug fixes

- In addition to the new synapse creation syntax, sparse probabilistic connections are now created much faster.

- The time for the initialization phase at the beginning of a `run()` has been significantly reduced.

- Multicompartmental simulations with a large number of compartments are now simulated more efficiently and are making better use of several processor cores when OpenMP is activated in C++ standalone mode. Thanks to Moritz Augustin for this contribution.

- Simulations will use compiler settings that optimize performance by default.

- Objects that have user-specified names are better supported for complex simulation scenarios (names no longer have to be unique at all times, but only across a network or across a standalone device).

- Various fixes for compatibility with recent versions of numpy and sympy

Important backwards-incompatible changes

- The argument names in `Synapses.connect` have changed and the first argument can no longer be an array of indices. To connect based on indices, use `Synapses.connect(i=source_indices, j=target_indices)`. See *Creating synapses* and the documentation of `Synapses.connect` for more details.

- The actions triggered by pre-synaptic and post-synaptic spikes are now described by the `on_pre` and `on_post` keyword arguments (instead of `pre` and `post`).

- The `Morphology` object no longer allows to change attributes such as length and diameter after its creation. Complex morphologies should instead be created using the `Section` class, allowing for the specification of all details.

- *Morphology* objects that are defined with coordinates need to provide the start point (relative to the end point of the parent compartment) as the first coordinate. See *Creating a neuron morphology* for more details.

- For simulations using the C++ standalone mode, no longer call `Device.build` (if using a single `run()` call), or use `set_device()` with `build_on_run=False` (see *Standalone code generation*).

Infrastructure improvements

- Our test suite is now also run on Mac OS-X (on the Travis CI platform).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Moritz Augustin (@moritzaugustin)

- Jan-Hendrik Schleimer (@ttxtea)

- Romain Cazé (@rcaze)

- Konrad Wartke (@Kwartke)

- Romain Brette (@romainbrette)

Testing, suggestions and bug reports (ordered alphabetically, apologies to anyone we forgot…):

- Chaofei Hong

- Kees de Leeuw

- Luke Y Prince

- Myung Seok Shim

- Owen Mackwood

- Github users: @epaxon, @flinz, @mariomulansky, @martinosorb, @neuralyzer, @oleskiw, @prcastro, @su-doankit

1.3.24 Brian 2.0b4

This is the fourth (and probably last) beta release for Brian 2.0. This release adds a few important new features and fixes a number of bugs so we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1. Note that the new recommended way to install Brian 2 is to use the Anaconda distribution and to install the Brian 2 conda package (see *Installation*).

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- In addition to the standard threshold/reset, groups can now define "custom events". These can be recorded with the new *EventMonitor* (a generalization of *SpikeMonitor*) and *Synapses* can connect to these events instead of the standard spike event. See *Custom events* for more details.

- *SpikeMonitor* and *EventMonitor* can now also record state variable values at the time of spikes (or custom events), thereby offering the functionality of StateSpikeMonitor from Brian 1. See *Recording variables at spike time* for more details.

- The code generation modes that interact with C++ code (weave, Cython, and C++ standalone) can now be more easily configured to work with external libraries (compiler and linker options, header files, etc.). See the documentation of the *cpp_prefs* module for more details.

Improvemements and bug fixes

- Cython simulations no longer interfere with each other when run in parallel (thanks to Daniel Bliss for reporting and fixing this).

- The C++ standalone now works with scalar delays and the spike queue implementation deals more efficiently with them in general.

- Dynamic arrays are now resized more efficiently, leading to faster monitors in runtime mode.

- The spikes generated by a *SpikeGeneratorGroup* can now be changed between runs using the *set_spikes* method.

- Multi-step state updaters now work correctly for non-autonomous differential equations

- *PoissonInput* now correctly works with multiple clocks (thanks to Daniel Bliss for reporting and fixing this)

- The get_states method now works for *StateMonitor*. This method provides a convenient way to access all the data stored in the monitor, e.g. in order to store it on disk.

- C++ compilation is now easier to get to work under Windows, see *Installation* for details.

Important backwards-incompatible changes

- The custom_operation method has been renamed to *run_regularly* and can now be called without the need for storing its return value.

- *StateMonitor* will now by default record at the beginning of a time step instead of at the end. See *Recording variables continuously* for details.

- Scalar quantities now behave as python scalars with respect to in-place modifications (augmented assignments). This means that x = 3*mV; y = x; y += 1*mV will no longer increase the value of the variable x as well.

Infrastructure improvements

- We now provide conda packages for Brian 2, making it very easy to install when using the Anaconda distribution (see *Installation*).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg ([@mstimberg](#))
- Dan Goodman ([@thesamovar](#))
- Daniel Bliss ([@dabliss](#))
- Romain Brette ([@romainbrette](#))

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot…):

- Daniel Bliss
- Damien Drix
- Rainer Engelken
- Beatriz Herrera Figueredo
- Owen Mackwood
- Augustine Tan
- Ot de Wiljes

1.3.25 Brian 2.0b3

This is the third beta release for Brian 2.0. This release does not add many new features but it fixes a number of important bugs so we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- A new `PoissonInput` class for efficient simulation of Poisson-distributed input events.

Improvements

- The order of execution for `pre` and `post` statements happenning in the same time step was not well defined (it fell back to the default alphabetical ordering, executing `post` before `pre`). It now explicitly specifies the `order` attribute so that `pre` gets executed before `post` (as in Brian 1). See the *Synapses* documentation for details.

- The default schedule that is used can now be set via a preference (*core.network.default_schedule*). New automatically generated scheduling slots relative to the explicitly defined ones can be used, e.g. `before_resets` or `after_synapses`. See *Scheduling* for details.

- The scipy package is no longer a dependency (note that weave for compiled C code under Python 2 is now available in a separate package). Note that multicompartmental models will still benefit from the scipy package if they are simulated in pure Python (i.e. with the `numpy` code generation target) – otherwise Brian 2 will fall back to a numpy-only solution which is significantly slower.

Important bug fixes

- Fix *SpikeGeneratorGroup* which did not emit all the spikes under certain conditions for some code generation targets ([#429](#))

- Fix an incorrect update of pre-synaptic variables in synaptic statements for the `numpy` code generation target ([#435](#)).

- Fix the possibility of an incorrect memory access when recording a subgroup with *SpikeMonitor* ([#454](#)).

- Fix the storing of results on disk for C++ standalone on Windows – variables that had the same name when ignoring case (e.g. `i` and `I`) where overwriting each other ([#455](#)).

Infrastructure improvements

- Brian 2 now has a chat room on [gitter](#): https://gitter.im/brian-team/brian2

- The sphinx documentation can now be built from the release archive file

- After a big cleanup, all files in the repository have now simple LF line endings (see https://help.github.com/articles/dealing-with-line-endings/ on how to configure your own machine properly if you want to contribute to Brian).

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg ([@mstimberg](#))

- Dan Goodman ([@thesamovar](#))

- Konrad Wartke ([@kwartke](#))

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot…):

- Daniel Bliss

- Owen Mackwood

- Ankur Sinha

- Richard Tomsett

1.3.26 Brian 2.0b2

This is the second beta release for Brian 2.0, we recommend all users of Brian 2 to upgrade. If you are a user new to Brian, we also recommend to directly start with Brian 2 instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- Multi-compartmental simulations can now be run using the *Standalone code generation* mode (this is not yet well-tested, though).

- The implementation of `TimedArray` now supports two-dimensional arrays, i.e. different input per neuron (or synapse, etc.), see *Timed arrays* for details.

- Previously, not setting a code generation target (using the *codegen.target* preference) would mean that the `numpy` target was used. Now, the default target is `auto`, which means that a compiled language (`weave` or `cython`) will be used if possible. See *Computational methods and efficiency* for details.

- The implementation of `SpikeGeneratorGroup` has been improved and it now supports a `period` argument to repeatedly generate a spike pattern.

Improvements

- The selection of a numerical algorithm (if none has been specified by the user) has been simplified. See *Numerical integration* for details.

- Expressions that are shared among neurons/synapses are now updated only once instead of for every neuron/synapse which can lead to performance improvements.

- On Windows, The Microsoft Visual C compiler is now supported in the `cpp_standalone` mode, see the respective notes in the *Installation* and *Computational methods and efficiency* documents.

- Simulation runs (using the standard "runtime" device) now collect profiling information. See *Profiling* for details.

Infrastructure and documentation improvements

- *Tutorials for beginners* in the form of ipython notebooks (currently only covering the basics of neurons and synapses) are now available.

- The *Examples* in the documentation now include the images they generated. Several examples have been adapted from Brian 1.

- The code is now automatically tested on Windows machines, using the appveyor service. This complements the Linux testing on travis.

- Using a version of a dependency (e.g. sympy) that we don't support will now raise an error when you import `brian2` – see *Dependency checks* for more details.

- Test coverage for the `cpp_standalone` mode has been significantly increased.

Important bug fixes

- The preparation time for complicated equations has been significantly reduced.

- The string representation of small physical quantities has been corrected (#361)

- Linking variables from a group of size 1 now works correctly (#383)

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)
- Dan Goodman (@thesamovar)
- Romain Brette (@romainbrette)
- Pierre Yger (@yger)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot…):

- Conor Cox
- Gordon Erlebacher
- Konstantin Mergenthaler

1.3.27 Brian 2.0beta

This is the first beta release for Brian 2.0 and the first version of Brian 2.0 we recommend for general use. From now on, we will try to keep changes that break existing code to a minimum. If you are a user new to Brian, we'd recommend to start with the Brian 2 beta instead of using the stable release of Brian 1.

This is however still a Beta release, please report bugs or suggestions to the github bug tracker (https://github.com/brian-team/brian2/issues) or to the brian-development mailing list (brian-development@googlegroups.com).

Major new features

- New classes *Morphology* and *SpatialNeuron* for the simulation of *Multicompartment models*
- A temporary "bridge" for `brian.hears` that allows to use its Brian 1 version from Brian 2 (*Brian Hears*)
- Cython is now a new code generation target, therefore the performance benefits of compiled code are now also available to users running simulations under Python 3.x (where `scipy.weave` is not available)
- Networks can now store their current state and return to it at a later time, e.g. for simulating multiple trials starting from a fixed network state (*Continuing/repeating simulations*)
- C++ standalone mode: multiple processors are now supported via OpenMP (*Multi-threading with OpenMP*), although this code has not yet been well tested so may be inaccurate.
- C++ standalone mode: after a run, state variables and monitored values can be loaded from disk transparently. Most scripts therefore only need two additional lines to use standalone mode instead of Brian's default runtime mode (*Standalone code generation*).

Syntax changes

- The syntax and semantics of everything around simulation time steps, clocks, and multiple runs have been cleaned up, making `reinit` obsolete and also making it unnecessary for most users to explicitly generate *Clock* objects – instead, a `dt` keyword can be specified for objects such as *NeuronGroup* (*Running a simulation*)
- The `scalar` flag for parameters/subexpressions has been renamed to `shared`
- The "unit" for boolean variables has been renamed from `bool` to `boolean`
- C++ standalone: several keywords of *CPPStandaloneDevice.build* have been renamed

---

- The preferences are now accessible via `prefs` instead of `brian_prefs`

- The `runner` method has been renamed to *custom_operation*

Improvements

- Variables can now be linked across `NeuronGroup`s (*Linked variables*)

- More flexible progress reporting system, progress reporting also works in the C++ standalone mode (*Progress reporting*)

- State variables can be declared as `integer` (*Equation strings*)

Bug fixes

57 github issues have been closed since the alpha release, of which 26 had been labeled as bugs. We recommend all users of Brian 2 to upgrade.

Contributions

Code and documentation contributions (ordered by the number of commits):

- Marcel Stimberg (@mstimberg)

- Dan Goodman (@thesamovar)

- Romain Brette (@romainbrette)

- Pierre Yger (@yger)

- Werner Beroux (@wernight)

Testing, suggestions and bug reports (ordered alphabetically, apologies to everyone we forgot…):

- Guillaume Bellec

- Victor Benichoux

- Laureline Logiaco

- Konstantin Mergenthaler

- Maurizio De Pitta

- Jan-Hendrick Schleimer

- Douglas Sterling

- Katharina Wilmes

1.4 Changes for Brian 1 users

- *Physical units*
- *Unported packages*
- *Replacement packages*
- *Removed classes/functions and their replacements*

In most cases, Brian 2 works in a very similar way to Brian 1 but there are some important differences to be aware of. The major distinction is that in Brian 2 you need to be more explicit about the definition of your simulation in order to avoid inadvertent errors. In some cases, you will now get a warning in other even an error – often the error/warning message describes a way to resolve the issue.

Specific examples how to convert code from Brian 1 can be found in the document *Detailed Brian 1 to Brian 2 conversion notes*.

1.4.1 Physical units

The unit system now extends to arrays, e.g. `np.arange(5) * mV` will retain the units of volts and not discard them as Brian 1 did. Brian 2 is therefore also more strict in checking the units. For example, if the state variable `v` uses the unit of volt, the statement `G.v = np.rand(len(G)) / 1000.` will now raise an error. For consistency, units are returned everywhere, e.g. in monitors. If `mon` records a state variable v, `mon.t` will return a time in seconds and `mon.v` the stored values of `v` in units of volts.

If you need a pure numpy array without units for further processing, there are several options: if it is a state variable or a recorded variable in a monitor, appending an underscore will refer to the variable values without units, e.g. `mon.t_` returns pure floating point values. Alternatively, you can remove units by diving by the unit (e.g. `mon.t / second`) or by explicitly converting it (`np.asarray(mon.t)`).

Here's an overview showing a few expressions and their respective values in Brian 1 and Brian 2:

Expression	Brian 1	Brian 2
1 * mV	1.0 * mvolt	1.0 * mvolt
np.array(1) * mV	0.001	1.0 * mvolt
np.array([1]) * mV	array([0.001])	array([1.]) * mvolt
np.mean(np.arange(5) * mV)	0.002	2.0 * mvolt
np.arange(2) * mV	array([0. , 0.001])	array([0., 1.]) * mvolt
(np.arange(2) * mV) >= 1 * mV	array([False, True], dtype=bool)	array([False, True], dtype=bool)
(np.arange(2) * mV)[0] >= 1 * mV	False	False
(np.arange(2) * mV)[1] >= 1 * mV	DimensionMismatchError	True

1.4.2 Unported packages

The following packages have not (yet) been ported to Brian 2. If your simulation critically depends on them, you should consider staying with Brian 1 for now.

- `brian.tools`

- `brian.library.modelfitting`

- `brian.library.electrophysiology`

1.4.3 Replacement packages

The following packages that were included in Brian 1 have now been split into separate packages.

- `brian.hears` has been updated to brian2hears. Note that there is a legacy package `brian2.hears` included in `brian2`, but this is now deprecated and will be removed in a future release. For now, see *Brian Hears* for details.

1.4.4 Removed classes/functions and their replacements

In Brian 2, we have tried to keep the number of classes/functions to a minimum, but make each of them flexible enough to encompass a large number of use cases. A lot of the classes and functions that existed in Brian 1 have therefore been removed. The following table lists (most of) the classes that existed in Brian 1 but do no longer exist in Brian 2. You can consult it when you get a `NameError` while converting an existing script from Brian 1. The third column links to a document with further explanation and the second column gives either:

1. the equivalent class in Brian 2 (e.g. *StateMonitor* can record multiple variables now and therefore replaces `MultiStateMonitor`);

2. the name of a Brian 2 class in square brackets (e.g. [*Synapses*] for `STDP`), this means that the class can be used as a replacement but needs some additional code (e.g. explicitly specified STDP equations). The "More details" document should help you in making the necessary changes;

3. "string expression", if the functionality of a previously existing class can be expressed using the general string expression framework (e.g. `threshold=VariableThreshold('Vt', 'V')` can be replaced by `threshold='V > Vt'`);

4. a link to the relevant github issue if no equivalent class/function does exist so far in Brian 2;

5. a remark such as "obsolete" if the particular class/function is no longer needed.

Brian 1	Brian 2	More details
AdEx	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
aEIF	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
AERSpikeMonitor	#298	*Monitors (Brian 1 –> 2 conversion)*
alpha_conductance	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
alpha_current	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
alpha_synapse	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
AutoCorrelogram	[*SpikeMonitor*]	*Monitors (Brian 1 –> 2 conversion)*
biexpr_conductance	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
biexpr_current	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
biexpr_synapse	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
Brette_Gerstner	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
CoincidenceCounter	[*SpikeMonitor*]	*Monitors (Brian 1 –> 2 conversion)*

Table 1 – continued from previous page

Brian 1	Brian 2	More details
`CoincidenceMatrixCounter`	[*SpikeMonitor*]	*Monitors (Brian 1 –> 2 conversion)*
`Compartments`	#443	*Multicompartmental models (Brian 1 –> 2 conversion)*
`Connection`	*Synapses*	*Synapses (Brian 1 –> 2 conversion)*
`Current`	#443	*Multicompartmental models (Brian 1 –> 2 conversion)*
`CustomRefractoriness`	[string expression]	*Neural models (Brian 1 –> 2 conversion)*
`DefaultClock`	*Clock*	*Networks and clocks (Brian 1 –> 2 conversion)*
`EmpiricalThreshold`	string expression	*Neural models (Brian 1 –> 2 conversion)*
`EventClock`	*Clock*	*Networks and clocks (Brian 1 –> 2 conversion)*
`exp_conductance`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`exp_current`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`exp_IF`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`exp_synapse`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`FileSpikeMonitor`	#298	*Monitors (Brian 1 –> 2 conversion)*
`FloatClock`	*Clock*	*Networks and clocks (Brian 1 –> 2 conversion)*
`FunReset`	[string expression]	*Neural models (Brian 1 –> 2 conversion)*
`FunThreshold`	[string expression]	*Neural models (Brian 1 –> 2 conversion)*
`hist_plot`	no equivalent	–
`HomogeneousPoissonThreshold`	string expression	*Neural models (Brian 1 –> 2 conversion)*
`IdentityConnection`	*Synapses*	*Synapses (Brian 1 –> 2 conversion)*
`IonicCurrent`	#443	*Multicompartmental models (Brian 1 –> 2 conversion)*
`ISIHistogramMonitor`	[*SpikeMonitor*]	*Monitors (Brian 1 –> 2 conversion)*
`Izhikevich`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`K_current_HH`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`leak_current`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`leaky_IF`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`MembraneEquation`	#443	*Multicompartmental models (Brian 1 –> 2 conversion)*
`MultiStateMonitor`	*StateMonitor*	*Monitors (Brian 1 –> 2 conversion)*
`Na_current_HH`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`NaiveClock`	*Clock*	*Networks and clocks (Brian 1 –> 2 conversion)*
`NoReset`	obsolete	*Neural models (Brian 1 –> 2 conversion)*
`NoThreshold`	obsolete	*Neural models (Brian 1 –> 2 conversion)*
`OfflinePoissonGroup`	[*SpikeGeneratorGroup*]	*Inputs (Brian 1 –> 2 conversion)*
`OrnsteinUhlenbeck`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`perfect_IF`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`PoissonThreshold`	string expression	*Neural models (Brian 1 –> 2 conversion)*
`PopulationSpikeCounter`	*SpikeMonitor*	*Monitors (Brian 1 –> 2 conversion)*
`PulsePacket`	[*SpikeGeneratorGroup*]	*Inputs (Brian 1 –> 2 conversion)*
`quadratic_IF`	[*Equations*]	*Library models (Brian 1 –> 2 conversion)*
`raster_plot`	`plot_raster`(brian2tools)	brian2tools documentation
`RecentStateMonitor`	no direct equivalent	*Monitors (Brian 1 –> 2 conversion)*
`Refractoriness`	string expression	*Neural models (Brian 1 –> 2 conversion)*
`RegularClock`	*Clock*	*Networks and clocks (Brian 1 –> 2 conversion)*
`Reset`	string expression	*Neural models (Brian 1 –> 2 conversion)*
`SimpleCustomRefractoriness`	[string expression]	*Neural models (Brian 1 –> 2 conversion)*
`SimpleFunThreshold`	[string expression]	*Neural models (Brian 1 –> 2 conversion)*
`SpikeCounter`	*SpikeMonitor*	*Monitors (Brian 1 –> 2 conversion)*
`StateHistogramMonitor`	[*StateMonitor*]	*Monitors (Brian 1 –> 2 conversion)*
`StateSpikeMonitor`	*SpikeMonitor*	*Monitors (Brian 1 –> 2 conversion)*
`STDP`	[*Synapses*]	*Synapses (Brian 1 –> 2 conversion)*

Table 1 – continued from previous page

Brian 1	Brian 2	More details
STP	[*Synapses*]	*Synapses (Brian 1 –> 2 conversion)*
StringReset	string expression	*Neural models (Brian 1 –> 2 conversion)*
StringThreshold	string expression	*Neural models (Brian 1 –> 2 conversion)*
Threshold	string expression	*Neural models (Brian 1 –> 2 conversion)*
VanRossumMetric	[*SpikeMonitor*]	*Monitors (Brian 1 –> 2 conversion)*
VariableReset	string expression	*Neural models (Brian 1 –> 2 conversion)*
VariableThreshold	string expression	*Neural models (Brian 1 –> 2 conversion)*

List of detailed instructions

Detailed Brian 1 to Brian 2 conversion notes

These documents are only relevant for former users of Brian 1. If you do not have any Brian 1 code to convert, go directly to the main *User's guide*.

Neural models (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about defining neural models, see the document *Models and neuron groups*.

> - *Threshold and Reset*
> - *Refractoriness*
> - *Subgroups*
> - *Linked Variables*

The syntax for specifying neuron models in a `NeuronGroup` changed in several details. In general, a string-based syntax (that was already optional in Brian 1) consistently replaces the use of classes (e.g. `VariableThreshold`) or guessing (e.g. which variable does `threshold=50*mV` check).

Threshold and Reset

String-based thresholds are now the only possible option and replace all the methods of defining threshold/reset in Brian 1:

Brian 1	Brian 2

``
group = NeuronGroup(N, 'dv/dt = -v / tau␣
↪: volt',
                threshold=-50*mV,
                reset=-70*mV)
``
	``
	group = NeuronGroup(N, 'dv/dt = -v / tau␣
	↪: volt',
	                        threshold='v > -50*mV
	↪',
	                        reset='v = -70*mV')
	``

continues on next page

Table 2 – continued from previous page

Brian 1	Brian 2
| ```python
group = NeuronGroup(N, 'dv/dt = -v / tau␣
↪: volt',
                  threshold=Threshold(-
↪50*mV, state='v'),
                  reset=Reset(-70*mV,␣
↪state='w'))
``` | ```python
group = NeuronGroup(N, 'dv/dt = -v / tau␣
↪: volt',
                  threshold='v > -50*mV
↪',
                  reset='v = -70*mV')
``` |
| ```python
group = NeuronGroup(N, '''dv/dt = -v /␣
↪tau : volt
                  dvt/dt = -vt /␣
↪tau : volt
                  vr : volt''',
                  ␣
↪threshold=VariableThreshold(state='v',

↪       threshold_state='vt'),
                  ␣
↪reset=VariableThreshold(state='v',

↪   resetvaluestate='vr'))
``` | ```python
group = NeuronGroup(N, '''dv/dt = -v /␣
↪tau : volt
                  dvt/dt = -vt /␣
↪tau : volt
                  vr : volt''',
                  threshold='v > vt',
                  reset='v = vr')
``` |
| ```python
group = NeuronGroup(N, 'rate : Hz',
                  ␣
↪threshold=PoissonThreshold(state='rate
↪'))
``` | ```python
group = NeuronGroup(N, 'rate : Hz',
                  threshold='rand()
↪<rate*dt')
``` |

There's no direct equivalent for the "functional threshold/reset" mechanism from Brian 1. In simple cases, they can be implemented using the general string expression/statement mechanism (note that in Brian 1, `reset=myreset` is equivalent to `reset=FunReset(myreset)`):

Brian 1	Brian 2
```def myreset(P,spikes): P.v_[spikes] = -↪70*mV+rand(len(spikes))*5*mVgroup = NeuronGroup(N, 'dv/dt = -v / tau↪: volt', threshold=-50*mV, reset=myreset)```	```group = NeuronGroup(N, 'dv/dt = -v / tau↪: volt', threshold='v > -50*mV↪', reset='-70*mV +↪rand()*5*mV')```
```def mythreshold(v): return (v > -50*mV) & (rand(N) > 0.5)group = NeuronGroup(N, 'dv/dt = -v / tau↪: volt', ↪↪threshold=SimpleFunThreshold(mythreshold,↪↪ state='v'), reset=-70*mV)```	```group = NeuronGroup(N, 'dv/dt = -v / tau↪: volt', threshold='v > -↪50*mV and rand() > 0.5', reset='v = -70*mV')```

For more complicated cases, you can use the general mechanism for *User-provided functions* that Brian 2 provides. The only caveat is that you'd have to provide an implementation of the function in the code generation target language which is by default C++ or Cython. However, in the default *Runtime code generation* mode, you can chose different code generation targets for different parts of your simulation. You can thus switch the code generation target for the threshold/reset mechanism to `numpy` while leaving the default target for the rest of the simulation in place. The details of this process and the correct definition of the functions (e.g. `global_reset` needs a "dummy" return value) are somewhat cumbersome at the moment and we plan to make them more straightforward in the future. Also note that if you use this kind of mechanism extensively, you'll lose all the performance advantage that Brian 2's code generation mechanism provides (in addition to not being able to use *Standalone code generation* mode at all).

Brian 1	Brian 2
| ```python
def single_threshold(v):
    # Only let a single neuron spike
    crossed_threshold = np.nonzero(v > -
    50*mV)[0]
    should_spike = np.zeros(len(P),
    dtype=np.bool)
    if len(crossed_threshold):
        choose = np.random.
    randint(len(crossed_threshold))
        should_spike[crossed_
    threshold[choose]] = True
    return should_spike

def global_reset(P, spikes):
    # Reset everything
    if len(spikes):
        P.v_[:] = -70*mV

neurons = NeuronGroup(N, 'dv/dt = -v /
    tau : volt',
                    
    threshold=SimpleFunThreshold(single_
    threshold,
    
            state='v'),
                    reset=global_reset)
``` | ```python
@check_units(v=volt, result=bool)
def single_threshold(v):
    pass # ... (identical to Brian 1)

@check_units(spikes=1, result=1)
def global_reset(spikes):
    # Reset everything
    if len(spikes):
        neurons.v_[:] = -0.070

neurons = NeuronGroup(N, 'dv/dt = -v /
    tau : volt',
                        threshold='single_
    threshold(v)',
                        reset='dummy =
    global_reset(i)')
# Set the code generation target for
    threshold/reset only:
neuron.thresholder['spike'].codeobj_
    class = NumpyCodeObject
neuron.resetter['spike'].codeobj_class =
    NumpyCodeObject
``` |

For an example how to translate `EmpiricalThreshold`, see the section on "Refractoriness" below.

Refractoriness

For a detailed description of Brian 2's refractoriness mechanism see *Refractoriness*.

In Brian 1, refractoriness was tightly linked with the reset mechanism and some combinations of refractoriness and reset were not allowed. The standard refractory mechanism had two effects during the refractoriness: it prevented the refractory cell from spiking and it clamped a state variable (normally the membrane potential of the cell). In Brian 2, refractoriness is independent of reset and the two effects are specified separately: the `refractory` keyword specifies the time (or an expression evaluating to a time) during which the cell does not spike, and the `(unless refractory)` flag marks one or more variables to be clamped during the refractory period. To correctly translate the standard refractory mechanism from Brian 1, you'll therefore need to specify both:

Brian 1	Brian 2
| ```python
group = NeuronGroup(N, 'dv/dt = (I - v)/
    tau : volt',
                    threshold=-50*mV,
                    reset=-70*mV,
                    refractory=3*ms)
``` | ```python
group = NeuronGroup(N, 'dv/dt = (I - v)/
    tau : volt (unless refractory)',
                    threshold='v > -50*mV
    ',
                    reset='v = -70*mV',
                    refractory=3*ms)
``` |

More complex refractoriness mechanisms based on `SimpleCustomRefractoriness` and `CustomRefrac-`
`toriness` can be translatated using string expressions or user-defined functions, see the remarks in the preceding
section on "Threshold and Reset".

Brian 2 no longer has an equivalent to the `EmpiricalThreshold` class (which detects at the first threshold crossing
but ignores all following threshold crossings for a certain time after that). However, the standard refractoriness mechanism
can be used to implement the same behaviour, since it does not reset/clamp any value if not explicitly asked for it (which
would be fatal for Hodgkin-Huxley type models):

Brian 1	Brian 2
| ```
group = NeuronGroup(N,'''
              dv/dt = (I_L - I_Na -
↪ I_K + I)/Cm : volt
              ...''',
             ↵
↪threshold=EmpiricalThreshold(threshold=20 ↵
↪

↪       refractory=1*ms,
             ↵
↪       state='v'))
``` | ```
group = NeuronGroup(N,'''
              dv/dt = (I_L - I_Na -
↪ I_K + I)/Cm : volt
              ...''',
              threshold='v > -20*mV
↪',
              refractory=1*ms)
``` |

Subgroups

The class *NeuronGroup* in Brian 2 does no longer provide a `subgroup` method, the only way to construct subgroups
is therefore the slicing syntax (that works in the same way as in Brian 1):

Brian 1	Brian 2
| ```
group = NeuronGroup(4000, ...)
group_exc = group.subgroup(3200)
group_inh = group.subgroup(800)
``` | ```
group = NeuronGroup(4000, ...)
group_exc = group[:3200]
group_inh = group[3200:]
``` |

Linked Variables

For a description of Brian 2's mechanism to link variables between groups, see *Linked variables*.

Linked variables need to be explicitly annotated with the `(linked)` flag in Brian 2:

Brian 1	Brian 2
```group1 = NeuronGroup(N, 'dv/dt = -v / tau :↵ →volt') group2 = NeuronGroup(N, '''dv/dt = (-v + w)↵ →/ tau : volt w : volt''') group2.w = linked_var(group1, 'v')```	```group1 = NeuronGroup(N, 'dv/dt = -v / tau :↵ →volt') group2 = NeuronGroup(N, '''dv/dt = (-v + w)↵ →/ tau : volt w : volt (linked) →''') group2.w = linked_var(group1, 'v')```

Synapses (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about defining and creating synapses, see the document *Synapses*.

> - *Converting Brian 1's `Connection` class*
> - *Converting Brian 1's `Synapses` class*

Converting Brian 1's `Connection` class

In Brian 2, the *`Synapses`* class is the only class to model synaptic connections, you will therefore have to convert all uses of Brian 1's `Connection` class. The `Connection` class increases a post-synaptic variable by a certain amount (the "synaptic weight") each time a pre-synaptic spike arrives. This has to be explicitly specified when using the *`Synapses`* class, the equivalent to the basic `Connection` usage is:

Brian 1	Brian 2
```conn = Connection(source, target, 'ge')```	```conn = Synapses(source, target, 'w :↵ →siemens', on_pre='ge += w')```

Note that he variable `w`, which stores the synaptic weight, has to have the same units as the post-synaptic variable (in this case: `ge`) that it increases.

Creating synapses and setting weights

With the `Connection` class, creating a synapse and setting its weight is a single process whereas with the *Synapses* class those two steps are separate. There is no direct equivalent to the convenience functions `connect_full`, `connect_random` and `connect_one_to_one`, but you can easily implement the same functionality with the general mechanism of *Synapses.connect*:

Brian 1	Brian 2

```conn1 = Connection(source, target, 'ge')```
 ```conn1[3, 5] = 3*nS```
	```conn1 = Synapses(source, target, 'w:```
	 ```↪siemens',```
	 ``` on_pre='ge += w')```
	 ```conn1.connect(i=3, j=5)```
	 ```conn1.w[3, 5] = 3*nS # (or conn1.w =```
	 ```↪3*nS)```

```conn2 = Connection(source, target, 'ge')```
 ```conn2.connect_full(source, target, 5*nS)```
	```conn2 = ... # see above```
	 ```conn2.connect()```
	 ```conn2.w = 5*nS```

```conn3 = Connection(source, target, 'ge')```
 ```conn3.connect_random(source, target,```
 ```   sparseness=0.02,```
 ```   weight=2*ns)```
	```conn3 = ... # see above```
	 ```conn3.connect(p=0.02)```
	 ```conn3.w = 2*nS```

```conn4 = Connection(source, target, 'ge')```
 ```conn4.connect_one_to_one(source, target,```
 ```   weight=4*nS)```
	```conn4 = ... # see above```
	 ```conn4.connect(j='i')```
	 ```conn4.w = 4*nS```

```conn5 = IdentityConnection(source,```
 ```↪target,```
 ```   weight=3*nS)```
	```conn5 = Synapses(source, target,```
	 ``` 'w : siemens (shared)')```
	 ```conn5.w = 3*nS```

Weight matrices

Brian 2's *Synapses* class does not support setting the weights of a neuron with a weight matrix. However, *Synapses. connect* creates the synapses in a predictable order (first all synapses for the first pre-synaptic cell, then all synapses for the second pre-synaptic cell, etc.), so a reshaped "flat" weight matrix can be used:

Brian 1	Brian 2
```# len(source) == 20, len(target) == 30 conn6 = Connection(source, target, 'ge') W = rand(20, 30)*nS conn6.connect(source, target, weight=W)```	```# len(source) == 20, len(target) == 30 conn6 = Synapses(source, target, 'w:⏎ ↪siemens', on_pre='ge += w') W = rand(20, 30)*nS conn6.connect() conn6.w = W.flatten()```

However note that if your weight matrix can be described mathematically (e.g. random as in the example above), then you should not create a weight matrix in the first place but use Brian 2's mechanism to set variables based on mathematical expressions (in the above case: `conn5.w = 'rand()'`). Especially for big connection matrices this will have better performance, since it will be executed in generated code. You should only resort to explicit weight matrices when there is no alternative (e.g. to load weights from previous simulations).

In Brian 1, you can restrict the functions `connect`, `connect_random`, etc. to subgroups. Again, there is no direct equivalent to this in Brian 2, but the general string syntax allows you to make connections conditional on logical statements that refer to pre-/post-synaptic indices and can therefore also used to restrict the connection to a subgroup of cells. When you set the synaptic weights, you *can* however use subgroups to restrict the subset of weights you want to set.

Brian 1	Brian 2
```conn7 = Connection(source, target, 'ge') conn7.connect_full(source[:5],⏎ ↪target[5:10], 5*nS)```	```conn7 = Synapses(source, target, 'w:⏎ ↪siemens', on_pre='ge += w') conn7.connect('i < 5 and j >=5 and j <10 ↪') # Alternative (more efficient): # conn7.connect(j='k in range(5, 10) if⏎ ↪i < 5') conn7.w[source[:5], target[5:10]] = 5*nS```

Connections defined by functions

Brian 1 allowed you to pass in a function as the value for the weight argument in a `connect` call (and also for the sparseness argument in `connect_random`). You should be able to replace such use cases by the the general, string-expression based method:

Brian 1	Brian 2
| ```
conn8 = Connection(source, target, 'ge')
conn8.connect_full(source, target,
                   weight=lambda i,
↪j:(1+cos(i-j))*2*nS)
``` | ```
conn8 = Synapses(source, target, 'w:
↪siemens',
                      on_pre='ge += w')
conn8.connect()
conn8.w = '(1 + cos(i - j))*2*nS'
``` |
| ```
conn9 = Connection(source, target, 'ge')
conn9.connect_random(source, target,
                     sparseness=0.02,
                     ␣
↪weight=lambda:rand()*nS)
``` | ```
conn9 = ...  # see above
conn9.connect(p=0.02)
conn9.w = 'rand()*nS'
``` |
| ```
conn10 = Connection(source, target, 'ge')
conn10.connect_random(source, target,
                      sparseness=lambda␣
↪i,j:exp(-abs(i-j)*.1),
                      weight=2*ns)
``` | ```
conn10 = ...  # see above
conn10.connect(p='exp(-abs(i - j)*.1)')
conn10.w = 2*nS
``` |

Delays

The specification of delays changed in several aspects from Brian 1 to Brian 2: In Brian 1, delays where homogeneous by default, and heterogeneous delays had to be marked by `delay=True`, together with the specification of the maximum delay. In Brian 2, heterogeneous delays are the default and you do not have to state the maximum delay. Brian 1's syntax of specifying a pair of values to get randomly distributed delays in that range is no longer supported, instead use Brian 2's standard string syntax:

Brian 1	Brian 2
| ```
conn11 = Connection(source, target, 'ge',
↪ delay=True,
                    max_delay=5*ms)
conn11.connect_full(source, target,␣
↪weight=3*nS,
                    delay=(0*ms, 5*ms))
``` | ```
conn11 = Synapses(source, target, 'w :
↪siemens',
                      on_pre='ge += w')
conn11.connect()
conn11.w = 3*nS
conn11.delay = 'rand()*5*ms'
``` |

Modulation

In Brian 2, there's no need for the `modulation` keyword that Brian 1 offered, you can describe the modulation as part of the `on_pre` action:

Brian 1	Brian 2
```conn12 = Connection(source, target, 'ge', modulation='u')```	```conn12 = Synapses(source, target, 'w :␣ →siemens', on_pre='ge += w * u_pre ␣→')```

Structure

There's no equivalen for Brian 1's `structure` keyword in Brian 2, synapses are always stored in a sparse data structure. There is currently no support for changing synapses at run time (i.e. the "dynamic" structure of Brian 1).

Converting Brian 1's `Synapses` class

Brian 2's *Synapses* class works for the most part like the class of the same name in Brian 1. There are however some differences in details, listed below:

Synaptic models

The basic syntax to define a synaptic model is unchanged, but the keywords `pre` and `post` have been renamed to `on_pre` and `on_post`, respectively.

Brian 1	Brian 2
```stdp_syn = Synapses(inputs, neurons,␣ →model=''' w:1 dApre/dt = -Apre/ →taupre : 1 (event-driven) dApost/dt = -Apost/ →taupost : 1 (event-driven)''', pre='''ge + =w Apre += delta_ →Apre w = clip(w +␣ →Apost, 0, gmax)''', post='''Apost +=␣ →delta_Apost w = clip(w +␣ →Apre, 0, gmax)''')```	```stdp_syn = Synapses(inputs, neurons,␣ →model=''' w:1 dApre/dt = -Apre/ →taupre : 1 (event-driven) dApost/dt = -Apost/ →taupost : 1 (event-driven)''', on_pre='''ge + =w Apre += delta_ →Apre w = clip(w +␣ →Apost, 0, gmax)''', on_post='''Apost +=␣ →delta_Apost w = clip(w +␣ →Apre, 0, gmax)''')```

Lumped variables (summed variables)

The syntax to define lumped variables (we use the term "summed variables" in Brian 2) has been changed: instead of assigning the synaptic variable to the neuronal variable you'll have to include the summed variable in the synaptic equations with the flag (`summed`):

Brian 1	Brian 2

``
# a non-linear synapse (e.g. NMDA) 	
neurons = NeuronGroup(1, model='''
                dv/dt = (gtot - v)/
↪(10*ms) : 1
                gtot : 1''')
syn = Synapses(inputs, neurons,
         model='''
         dg/dt = -a*g+b*x*(1-g) : 1
         dx/dt = -c*x : 1
         w : 1 # synaptic weight''
↪',
         pre='x += w')
neurons.gtot=S.g
``
	``
	# a non-linear synapse (e.g. NMDA)
	neurons = NeuronGroup(1, model='''
	                dv/dt = (gtot - v)/
	↪(10*ms) : 1
	                gtot : 1''')
	syn = Synapses(inputs, neurons,
	         model='''
	         dg/dt = -a*g+b*x*(1-g) : 1
	         dx/dt = -c*x : 1
	         w : 1 # synaptic weight
	         gtot_post = g : 1 (summed)
	↪''',
	         on_pre='x += w')
	``

Creating synapses

In Brian 1, synapses were created by assigning `True` or an integer (the number of synapses) to an indexed *Synapses* object. In Brian 2, all synapse creation goes through the *Synapses.connect* function. For examples how to create more complex connection patterns, see the section on translating `Connections` objects above.

Brian 1	Brian 2

``
syn = Synapses(...)
# single synapse 	
syn[3, 5] = True
``
	``
	syn = Synapses(...)
	# single synapse
	syn.connect(i=3, j=5)
	``

``
# all-to-all connections 	
syn[:, :] = True
``
	``
	# all-to-all connections
	syn.connect()
	``

``
# all to neuron number 1 	
syn[:, 1] = True
``
	``
	# all to neuron number 1
	syn.connect(j='1')
	``

``
# multiple synapses 	
syn[4, 7] = 3
``
	``
	# multiple synapses
	syn.connect(i=4, j=7, n=3)
	``

``
# connection probability 2% 	
syn[:, :] = 0.02
``
	``
	# connection probability 2%
	syn.connect(p=0.02)
	``

Multiple pathways

As Brian 1, Brian 2 supports multiple pre- or post-synaptic pathways, with separate pre-/post-codes and delays. In Brian 1, you have to specify the pathways as tuples and can then later access them individually by using their index. In Brian 2, you specify the pathways as a dictionary, i.e. by giving them individual names which you can then later use to access them (the default pathways are called `pre` and `post`):

Brian 1	Brian 2
| ```
S = Synapses(...,
          pre=('ge + =w',
               '''w = clip(w + Apost,␣
→0, inf)
               Apre += delta_Apre''
→'),
          post='''Apost += delta_Apost
               w = clip(w + Apre,␣
→0, inf)''')

S[:, :] = True
S.delay[1][:, :] = 3*ms # delayed trace
``` | ```
S = Synapses(...,
          pre={'pre_transmission':
               'ge += w',
               'pre_plasticity':
               '''w = clip(w + Apost,␣
→0, inf)
               Apre += delta_Apre''
→'},
          post='''Apost += delta_Apost
               w = clip(w + Apre,␣
→0, inf)''')

S.connect()
S.pre_plasticity.delay[:, :] = 3*ms #␣
→delayed trace
``` |

Monitoring synaptic variables

Both in Brian 1 and Brian 2, you can record the values of synaptic variables with a *StateMonitor*. You no longer have to call an explicit indexing function, but you can directly provide an appropriately indexed *Synapses* object. You can now also use the same technique to index the *StateMonitor* object to get the recorded values, see the respective section in the *Synapses* documentation for details.

Brian 1	Brian 2
| ```
syn = Synapses(...)
# record all synapse targetting neuron 3
indices = syn.synapse_index((slice(None),
→ 3))
mon = StateMonitor(S, 'w',␣
→record=indices)
``` | ```
syn = Synapses(...)
# record all synapse targetting neuron 3
mon = StateMonitor(S, 'w', record=S[:,␣
→3])
``` |

Inputs (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about adding external stimulation to a network, see the document *Input stimuli*.

- *Poisson Input*

- *Spike generation*

- *Arbitrary time-dependent input (`TimedArray`)*

Poisson Input

Brian 2 provides the same two groups that Brian 1 provided: `PoissonGroup` and `PoissonInput`. The mechanism for inhomogeneous Poisson processes has changed: instead of providing a Python function of time, you'll now have to provide a string expression that is evaluated at every time step. For most use cases, this should allow a direct translation:

Brian 1	Brian 2
```python rates = lambda␣ ↪t:(1+cos(2*pi*t*1*Hz))*10*Hz group = PoissonGroup(100, rates=rates) ```	```python rates = '(1 + cos(2*pi*t*1*Hz)*10*Hz)' group = PoissonGroup(100, rates=rates) ```

For more complex rate modulations, the expression can refer to *User-provided functions* and/or you can replace the `PoissonGroup` by a general `NeuronGroup` with a threshold condition `rand()<rates*dt` (which allows you to store per-neuron attributes).

There is currently no direct replacement for the more advanced features of `PoissonInput` (`record`, `freeze`, `copies`, `jitter`, and `reliability` keywords), but various workarounds are possible, e.g. by directly using a `BinomialFunction` in the equations. For example, you can get the functionality of the `freeze` keyword (identical Poisson events for all neurons) by storing the input in a shared variable and then distribute the input to all neurons:

Brian 1	Brian 2
```python group = NeuronGroup(10, 'dv/dt = -v/(10*ms)␣ ↪: 1') input = PoissonInput(group, N=1000,␣ ↪rate=1*Hz, weight=0.1, state='v ↪', freeze=True) ```	```python group = NeuronGroup(10, '''dv/dt = -v /␣ ↪(10*ms) : 1 shared_input␣ ↪: 1 (shared)''') poisson_input = BinomialFunction(n=1000,␣ ↪p=1*Hz*group.dt) group.run_regularly('''shared_input =␣ ↪poisson_input()*0.1 v += shared_input' ↪'') ```

Spike generation

SpikeGeneratorGroup provides mostly the same functionality as in Brian 1. In contrast to Brian 1, there is only one way to specify which neurons spike and when – you have to provide the index array and the times array as separate arguments:

Brian 1	Brian 2
```gen1 = SpikeGeneratorGroup(2, [(0, 0*ms),↪ (1, 1*ms)])gen2 = SpikeGeneratorGroup(2, [(array([0,↪ 1]), 0*ms), (array([0,↪ 1]), 1*ms)]gen3 = SpikeGeneratorGroup(2, (array([0,↪1]), array([0,↪1])*ms))gen4 = SpikeGeneratorGroup(2, array([[0,↪0.0], [1,↪0.001]])```	```gen1 = SpikeGeneratorGroup(2, [0, 1], [0,↪ 1]*ms)gen2 = SpikeGeneratorGroup(2, [0, 1, 0,↪1], [0, 0, 1,↪1]*ms)gen3 = SpikeGeneratorGroup(2, [0, 1], [0,↪ 1]*ms)gen4 = SpikeGeneratorGroup(2, [0, 1], [0,↪ 1]*ms)```

Note: For large arrays, make sure to provide a *Quantity* array (e.g. [0, 1, 2]*ms) and not a list of *Quantity* values (e.g. [0*ms, 1*ms, 2*ms]). A list has first to be translated into an array which can take a considerable amount of time for a list with many elements.

There is no direct equivalent of the Brian 1 option to use a generator that updates spike times online. The easiest alternative in Brian 2 is to pre-calculate the spikes and then use a standard *SpikeGeneratorGroup*. If this is not possible (e.g. there are two many spikes to fit in memory), then you can workaround the restriction by using custom code (see *User-provided functions* and *Arbitrary Python code (network operations)*).

Arbitrary time-dependent input (`TimedArray`)

For a detailed description of the *TimedArray* mechanism in Brian 2, see *Timed arrays*.

In Brian 1, timed arrays where special objects that could be assigned to a state variable and would then be used to update this state variable at every time step. In Brian 2, a timed array is implemented using the standard *Functions* mechanism which has the advantage that more complex access patterns can be implemented (e.g. by not using t as an argument, but something like t - delay). This syntax was possible in Brian 1 as well, but was disadvantageous for performance and had other limits (e.g. no unit support, no linear integration). In Brian 2, these disadvantages no longer apply and the function syntax is therefore the only available syntax. You can convert the old-style Brian 1 syntax to Brian 2 as follows:

Warning: The example below does not correctly translate the changed semantics of *TimedArray* related to the time. In Brian 1, TimedArray([0, 1, 2], dt=10*ms) will return 0 for t<5*ms, 1 for 5*ms<=t<15*ms, and 2 for t>=15*ms. Brian 2 will return 0 for t<10*ms, 1 for 10*ms<=t<20*ms, and 2 for t>=20*ms.

Brian 1	Brian 2
``# same input for all neurons
eqs = '''
dv/dt = (I - v)/tau : volt
I : volt
'''
group = NeuronGroup(1, model=eqs,
reset=0*mV,␣
↪threshold=15*mV)
group.I = TimedArray(linspace(0*mV,␣
↪20*mV, 100),
dt=10*ms)```	```# same input for all neurons
I = TimedArray(linspace(0*mV, 20*mV,␣
↪100),
dt=10*ms)
eqs = '''
dv/dt = (I(t) - v)/tau : volt
'''
group = NeuronGroup(1, model=eqs,
reset='v = 0*mV',
threshold='v > 15*mV
↪')```
``# neuron-specific input
eqs = '''
dv/dt = (I - v)/tau : volt
I : volt
'''
group = NeuronGroup(5, model=eqs,
reset=0*mV,␣
↪threshold=15*mV)
values = (linspace(0*mV, 20*mV, 100)[:,␣
↪None] *
linspace(0, 1, 5))
group.I = TimedArray(values, dt=10*ms)```	```# neuron-specific input
values = (linspace(0*mV, 20*mV, 100)[:,␣
↪None] *
linspace(0, 1, 5))
I = TimedArray(values, dt=10*ms)
eqs = '''
dv/dt = (I(t, i) - v)/tau : volt
'''
group = NeuronGroup(5, model=eqs,
reset='v = 0*mV',
threshold='v > 15*mV
↪')```

Monitors (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about recording network activity, see the document *Recording during a simulation*.

> - *Monitoring spiking activity*
> - *Monitoring variables*

Monitoring spiking activity

The main class to record spiking activity is `SpikeMonitor` which is created in the same way as in Brian 1. However, the internal storage and retrieval of spikes is different. In Brian 1, spikes were stored as a list of pairs (i, t), the index and time of each spike. In Brian 2, spikes are stored as two arrays i and t, storing the indices and times. You can access these arrays as attributes of the monitor, there's also a convenience attribute it that returns both at the same time. The following table shows how the spike indices and times can be retrieved in various forms in Brian 1 and Brian 2:

Brian 1	Brian 2
| ```python
mon = SpikeMonitor(group)
#... do the run
list_of_pairs = mon.spikes
index_list, time_list = zip(*list_of_
↪pairs)
index_array = array(index_list)
time_array = array(time_list)
# time_array is unitless in Brian 1
``` | ```python
mon = SpikeMonitor(group)
#... do the run
list_of_pairs = zip(*mon.it)
index_list = list(mon.i)
time_list = list(mon.t)
index_array, time_array = mon.i, mon.t
# time_array has units in Brian 2
``` |

You can also access the spike times for individual neurons. In Brian 1, you could directly index the monitor which is no longer allowed in Brian 2. Instead, ask for a dictionary of spike times and index the returned dictionary:

Brian 1	Brian 2
| ```python
# dictionary of spike times for each␣
↪neuron:
spike_dict = mon.spiketimes
# all spikes for neuron 3:
spikes_3 = spike_dict[3] #  (no units)
spikes_3 = mon[3] #  alternative (no␣
↪units)
``` | ```python
# dictionary of spike times for each␣
↪neuron:
spike_dict = mon.spike_trains()
# all spikes for neuron 3:
spikes_3 = spike_dict[3]  # with units
``` |

In Brian 2, *SpikeMonitor* also provides the functionality of the Brian 1 classes `SpikeCounter` and `PopulationSpikeCounter`. If you are only interested in the counts and not in the individual spike events, use `record=False` to save the memory of storing them:

Brian 1	Brian 2
| ```python
counter = SpikeCounter(group)
pop_counter =␣
↪PopulationSpikeCounter(group)
#... do the run
# Number of spikes for neuron 3:
count_3 = counter[3]
# Total number of spikes:
total_spikes = pop_counter.nspikes
``` | ```python
counter = SpikeMonitor(group,␣
↪record=False)

#... do the run
# Number of spikes for neuron 3
count_3 = counter.count[3]
# Total number of spikes:
total_spikes = counter.num_spikes
``` |

Currently Brian 2 provides no functionality to calculate statistics such as correlations or histograms online, there is no equivalent to the following classes that existed in Brian 1: `AutoCorrelogram`, `CoincidenceCounter`, `CoincidenceMatrixCounter`, `ISIHistogramMonitor`, `VanRossumMetric`. You will therefore have to be calculate the corresponding statistiacs manually after the simulation based on the information stored in the *SpikeMonitor*. If you use the default *Runtime code generation*, you can also create a new Python class that calculates the statistic online (see this example from a Brian 2 tutorial).

Monitoring variables

Single variables are recorded with a *StateMonitor* in the same way as in Brian 1, but the times and variable values are accessed differently:

Brian 1	Brian 2
| ```python
mon = StateMonitor(group, 'v',
                  record=True)
# ... do the run
# plot the trace of neuron 3:
plot(mon.times/ms, mon[3]/mV)
# plot the traces of all neurons:
plot(mon.times/ms, mon.values.T/mV)
``` | ```python
mon = StateMonitor(group, 'v',
                        record=True)
# ... do the run
# plot the trace of neuron 3:
plot(mon.t/ms, mon[3].v/mV)
# plot the traces of all neurons:
plot(mon.t/ms, mon.v.T/mV)
``` |

Further differences:

- *StateMonitor* now records in the `'start'` scheduling slot by default. This leads to a more intuitive correspondence between the recorded times and the values: in Brian 1 (where *StateMonitor* recorded in the `'end'` slot) the recorded value at 0ms was not the initial value of the variable but the value after integrating it for a single time step. The disadvantage of this new default is that the very last value at the end of the last time step of a simulation is not recorded anymore. However, this value can be manually added to the monitor by calling *StateMonitor.record_single_timestep*.

- To not record every time step, use the `dt` argument (as for all other classes) instead of specifying a number of `timesteps`.

- Using `record=False` does no longer provide mean and variance of the recorded variable.

In contrast to Brian 1, *StateMonitor* can now record multiple variables and therefore replaces Brian 1's `MultiStateMonitor`:

Brian 1	Brian 2
| ```python
mon = MultiStateMonitor(group, ['v', 'w
 '],
                       record=True)
# ... do the run
# plot the traces of v and w for neuron
 3:
plot(mon['v'].times/ms, mon['v'][3]/mV)
plot(mon['w'].times/ms, mon['w'][3]/mV)
``` | ```python
mon = StateMonitor(group, ['v', 'w'],
                        record=True)
# ... do the run
# plot the traces of v and w for neuron
 3:
plot(mon.t/ms, mon[3].v/mV)
plot(mon.t/ms, mon[3].w/mV)
``` |

To record variable values at the times of spikes, Brian 2 no longer provides a separate class as Brian 1 did (StateSpikeMonitor). Instead, you can use *SpikeMonitor* to record additional variables (in addition to the neuron index and the spike time):

Brian 1	Brian 2
```# We assume that "group" has a varying↵ ↪threshold mon = StateSpikeMonitor(group, 'v') # ... do the run # plot the mean v at spike time for each↵ ↪neuron mean_values = [mean(mon.values('v', idx)) for idx in↵ ↪range(len(group))] plot(mean_values/mV, 'o')```	```# We assume that "group" has a varying↵ ↪threshold mon = SpikeMonitor(group, variables='v') # ... do the run # plot the mean v at spike time for each↵ ↪neuron values = mon.values('v') mean_values = [mean(values[idx]) for idx in↵ ↪range(len(group))] plot(mean_values/mV, 'o')```

Note that there is no equivalent to StateHistogramMonitor, you will have to calculate the histogram from the recorded values or write your own custom monitor class.

Networks and clocks (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about running simulations, controling the simulation timestep, etc., see the document *Running a simulation*.

> - *Clocks and timesteps*
> - *Networks*

Clocks and timesteps

Brian's system of handling clocks has substantially changed. For details about the new system in place see *Setting the simulation time step*. The main differences to Brian 1 are:

- There is no more "clock guessing" – objects either use the *defaultclock* or a dt/clock value that was explicitly specified during their construction.

- In Brian 2, the time step is allowed to change after the creation of an object and between runs – the relevant value is the value in place at the point of the *run()* call.

- It is rarely necessary to create an explicit *Clock* object, most of the time you should use the *defaultclock* or provide a dt argument during the construction of the object.

- There's only one *Clock* class, the (deprecated) FloatClock, RegularClock, etc. classes that Brian 1 provided no longer exist.

- It is no longer possible to (re-)set the time of a clock explicitly, there is no direct equivalent of Clock.reinit and reinit_default_clock. To start a completely new simulation after you have finished a previous one, either create a new *Network* or use the *start_scope()* mechanism. To "rewind" a simulation to a previous point, use the new *store()*/*restore()* mechanism. For more details, see below and *Running a simulation*.

Networks

Both Brian 1 and Brian 2 offer two ways to run a simulation: either by explicitly creating a `Network` object, or by using a `MagicNetwork`, i.e. a simple `run()` statement.

Explicit network

The mechanism to create explicit `Network` objects has not changed significantly from Brian 1 to Brian 2. However, creating a new `Network` will now also automatically reset the clock back to 0s, and stricter checks no longer allow the inclusion of the same object in multiple networks.

Brian 1	Brian 2
``group = ...
mon = ...
net = Network(group, mon)
net.run(1*ms)

reinit()
group = ...
mon = ...
net = Network(group, mon)
net.run(1*ms)``` | ```group = ...
mon = ...
net = Network(group, mon)
net.run(1*ms)

# new network starts at 0s
group = ...
mon = ...
net = Network(group, mon)
net.run(1*ms)``` |

"Magic" network

For most simple, "flat", scripts (see e.g. the *Examples*), the `run()` statement in Brian 2 automatically collects all the Brian objects (`NeuronGroup`, etc.) into a "magic" network in the same way as Brian 1 did. The logic behind this collection has changed, though, with important consequences for more complex simulation scripts: in Brian 1, the magic network includes all Brian objects that have been *created* in the same execution frame as the `run()` call. Objects that are created in other functions could be added using `magic_return` and `magic_register`. In Brian 2, the magic network contains all Brian objects that are *visible* in the same execution frame as the `run()` call. The advantage of the new system is that it is clearer what will be included in the network and there is no danger of including previously created, but no longer needed, objects in a simulation. E.g. in the following example, a common mistake in Brian 1 was to not include the `clear()`, which meant that each run not only simulated the current objects, but also all objects from previous loop iterations. Also, without the `reinit_default_clock()`, each run would start at the end time of the previous run. In Brian 2, this loop does not need any explicit clearing up, each `run()` will only simulate the object that it "sees" (`group1`, `group2`, `syn`, and `mon`) and start each simulation at 0s:

Brian 1	Brian 2
``for r in range(100):
    reinit_default_clock()
    clear()
    group1 = NeuronGroup(...)
    group2 = NeuronGroup(...)
    syn = Synapses(group1, group2, ...)
    mon = SpikeMonitor(group2)
    run(1*second)``` | ```for r in range(100):

    group1 = NeuronGroup(...)
    group2 = NeuronGroup(...)
    syn = Synapses(group1, group2, ...)
    mon = SpikeMonitor(group2)
    run(1*second)``` |

There is no replacement for the `magic_return` and `magic_register` functions. If the returned object is stored in a variable at the level of the *run()* call, then it is no longer necessary to use `magic_return`, as the returned object is "visible" at the level of the *run()* call:

Brian 1	Brian 2

<pre>@magic_return
def f():
    return PoissonGroup(100,␣
↪rates=100*Hz)
 	
pg = f() # needs magic_return
mon = SpikeMonitor(pg)
run(100*ms)</pre>
	<pre>def f():
	    return PoissonGroup(100,␣
	↪rates=100*Hz)
	
	pg = f() # is "visible" and will be␣
	↪included
	mon = SpikeMonitor(pg)
	run(100*ms)</pre>

The general recommendation is however: if your script is complex (multiple functions/files/classes) and you are not sure whether some objects will be included in the magic network, use an explicit *Network* object.

Note that one consequence of the "is visible" approach is that objects stored in containers (lists, dictionaries, …) will not be automatically included in Brian 2. Use an explicit *Network* object to get around this restriction:

Brian 1	Brian 2

<pre>groups = {'exc': NeuronGroup(...),
         'inh': NeuronGroup(...)}
...
 	
run(5*ms)</pre>
	<pre>groups = {'exc': NeuronGroup(...),
	         'inh': NeuronGroup(...)}
	...
	net = Network(groups)
	net.run(5*ms)</pre>

External constants

In Brian 2, external constants are taken from the surrounding namespace at the point of the *run()* call and not when the object is defined (for other ways to define the namespace, see *External variables*). This allows to easily change external constants between runs, in contrast to Brian 1 where the whether this worked or not depended on details of the model (e.g. whether linear integration was used):

Brian 1	Brian 2

<pre>tau = 10*ms
# to be sure that changes between runs␣ 	
→are taken into
# account, define "I" as a neuronal␣ 	
→parameter
group = NeuronGroup(10, '''dv/dt = (-v +␣
→I) / tau : 1
                        I : 1''')
group.v = linspace(0, 1, 10)
group.I = 0.0
mon = StateMonitor(group, 'v',␣
→record=True)
run(5*ms)
group.I = 0.5
run(5*ms)
group.I = 0.0
run(5*ms)</pre>
	<pre>tau = 10*ms
	
	# The value for I will be updated at␣
	→each run
	group = NeuronGroup(10, 'dv/dt = (-v +␣
	→I) / tau : 1')
	
	group.v = linspace(0, 1, 10)
	I = 0.0
	mon = StateMonitor(group, 'v',␣
	→record=True)
	run(5*ms)
	I = 0.5
	run(5*ms)
	I = 0.0
	run(5*ms)</pre>

Preferences (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> For the main documentation about preferences, see the document *Preferences*.

In Brian 1, preferences were set either with the function `set_global_preferences` or by creating a module somewhere on the Python path called `brian_global_config.py`.

Setting preferences

The function `set_global_preferences` no longer exists in Brian 2. Instead, importing from `brian2` gives you a variable *prefs* that can be used to set preferences. For example, in Brian 1 you would write:

```
set_global_preferences(weavecompiler='gcc')
```

In Brian 2 you would write:

```
prefs.codegen.cpp.compiler = 'gcc'
```

Configuration file

The module `brian_global_config.py` is not used by Brian 2, instead we search for configuration files in the current directory, user directory or installation directory. In Brian you would have a configuration file that looks like this:

```python
from brian.globalprefs import *
set_global_preferences(weavecompiler='gcc')
```

In Brian 2 you would have a file like this:

```
codegen.cpp.compiler = 'gcc'
```

Preference name changes

- `defaultclock`: removed because it led to unclear behaviour of scripts.

- `useweave_linear_diffeq`: removed because it was no longer relevant.

- `useweave`: now replaced by *codegen.target* (but note that weave is no longer supported in Brian 2, use Cython instead).

- `weavecompiler`: now replaced by *codegen.cpp.compiler*.

- `gcc_options`: now replaced by *codegen.cpp.extra_compile_args_gcc*.

- `openmp`: now replaced by *devices.cpp_standalone.openmp_threads*.

- `usecodegen*`: removed because it was no longer relevant.

- `usenewpropagate`: removed because it was no longer relevant.

- `usecstdp`: removed because it was no longer relevant.

- `brianhears_usegpu`: removed because Brian Hears doesn't exist in Brian 2.

- `magic_useframes`: removed because it was no longer relevant.

Multicompartmental models (Brian 1 –> 2 conversion)

> **Brian 2 documentation**
>
> Support for multicompartmental models is now an integral part of Brian 2 (an early version of it was included as an experimental module in Brian 1). See the document *Multicompartment models*.

Brian 1 offered support for simple multi-compartmental models in the `compartments` module. This module allowed you to combine the equations for several compartments into a single `Equations` object. This is only a suitable solution for simple morphologies (e.g. "ball-and-stick" models) but has the advantage over using `SpatialNeuron` that you can have several of such neurons in a `NeuronGroup`.

If you already have a definition of a model using Brian 1's `compartments` module, then you can simply print out the equations and use them directly in Brian 2. For simple models, writing the equations without that help is rather straightforward anyway:

Brian 1	Brian 2

<pre>V0 = 10*mV
C = 200*pF
Ra = 150*kohm
R = 50*Mohm
soma_eqs = (MembraneEquation(C) +
          IonicCurrent('I=(vm-V0)/R :␣
→amp'))
dend_eqs = MembraneEquation(C)
neuron_eqs = Compartments({'soma': soma_
→eqs,
                      'dend': dend_
→eqs})
 	
neuron = NeuronGroup(N, neuron_eqs)</pre>
	<pre>V0 = 10*mV
	C = 200*pF
	Ra = 150*kohm
	R = 50*Mohm
	neuron_eqs = '''
	dvm_soma/dt = (I_soma + I_soma_dend)/C :␣
	→volt
	I_soma = (V0 - vm_soma)/R : amp
	I_soma_dend = (vm_dend - vm_soma)/Ra :␣
	→amp
	dvm_dend/dt = -I_soma_dend/C : volt'''
	
	neuron = NeuronGroup(N, neuron_eqs)</pre>

Library models (Brian 1 –> 2 conversion)

- *Neuron models*
- *Ionic currents*
- *Synapses*

Neuron models

The neuron models in Brian 1's `brian.library.IF` package are nothing more than shorthands for equations. The following table shows how the models from Brian 1 can be converted to explicit equations (and reset statements in the case of the adaptive exponential integrate-and-fire model) for use in Brian 2. The examples include a "current" `I` (depending on the model not necessarily in units of Ampère) and could e.g. be used to plot the f-I curve of the neuron.

Perfect integrator

Brian 1	Brian 2

<pre>eqs = (perfect_IF(tau=10*ms) +
      Current('I : volt'))
group = NeuronGroup(N, eqs,
              threshold='v > -50*mV
→',
              reset='v = -70*mV')</pre>
	<pre>tau = 10*ms
	eqs = '''dvm/dt = I/tau : volt
	        I : volt'''
	group = NeuronGroup(N, eqs,
	              threshold='v > -50*mV
	→',
	              reset='v = -70*mV')</pre>

Leaky integrate-and-fire neuron

Brian 1	Brian 2
| ```
eqs = (leaky_IF(tau=10*ms, El=-70*mV) +
       Current('I : volt'))
group = ... # see above
``` | ```
tau = 10*ms; El = -70*mV
eqs = '''dvm/dt = ((El - vm) + I)/tau :␣
→volt
        I : volt'''
group = ... # see above
``` |

Exponential integrate-and-fire neuron

Brian 1	Brian 2
| ```
eqs = (exp_IF(C=1*nF, gL=30*nS, EL=-
→70*mV,
          VT=-50*mV, DeltaT=2*mV) +
       Current('I : amp'))
group = ... # see above
``` | ```
C = 1*nF; gL = 30*nS; EL = -70*mV; VT = -
→50*mV; DeltaT = 2*mV
eqs = '''dvm/dt = (gL*(EL-
→vm)+gL*DeltaT*exp((vm-VT)/DeltaT) + I)/
→C : volt
        I : amp'''
group = ... # see above
``` |

Quadratic integrate-and-fire neuron

Brian 1	Brian 2
| ```
eqs = (quadratic_IF(C=1*nF, a=5*nS/mV,
       EL=-70*mV, VT=-50*mV) +
       Current('I : amp'))
group = ... # see above
``` | ```
C = 1*nF; a=5*nS/mV; EL=-70*mV; VT = -
→50*mV
eqs = '''dvm/dt = (a*(vm-EL)*(vm-VT) +␣
→I)/C : volt
        I : amp'''
group = ... # see above
``` |

Izhikevich neuron

Brian 1	Brian 2

``python
eqs = (Izhikevich(a=0.02/ms, b=0.2/ms) +
      Current('I : volt/second'))
group = ... # see above
``
	``python
	a = 0.02/ms; b = 0.2/ms
	eqs = '''dvm/dt = (0.04/ms/mV)*vm**2+(5/
	→ms)*vm+140*mV/ms-w + I : volt
	        dw/dt = a*(b*vm-w) : volt/second
	        I : volt/second'''
	group = ... # see above
	``

Adaptive exponential integrate-and-fire neuron ("Brette-Gerstner model")

Brian 1	Brian 2

``python
# AdEx, aEIF, and Brette_Gerstner all␣ 	
→refer to the same model
eqs = (aEIF(C=1*nF, gL=30*nS, EL=-70*mV,
          VT=-50*mV, DeltaT=2*mV,␣
→tauw=150*ms, a=4*nS) +
      Current('I:amp'))
group = NeuronGroup(N, eqs,
                    threshold='v > -20*mV
→',
                    ␣
→reset=AdaptiveReset(Vr=-70*mV, b=0.
→08*nA))
``
	``python
	C = 1*nF; gL = 30*nS; EL = -70*mV; VT = -
	→50*mV; DeltaT = 2*mV; tauw = 150*ms; a␣
	→= 4*nS
	eqs = '''dvm/dt = (gL*(EL-
	→vm)+gL*DeltaT*exp((vm-VT)/DeltaT) -w +␣
	→I)/C : volt
	        dw/dt=(a*(vm-EL)-w)/tauw : amp
	        I : amp'''
	group = NeuronGroup(N, eqs,
	                    threshold='vm > -
	→20*mV',
	                    reset='vm=-70*mV; w␣
	→+= 0.08*nA')
	``

Ionic currents

Brian 1's functions for ionic currents, provided in `brian.library.ionic_currents` correspond to the following equations (note that the currents follow the convention to use a shifted membrane potential, i.e. the membrane potential at rest is 0mV):

Brian 1	Brian 2
| ```python
from brian.library.ionic_currents import␣
↪*
defaultclock.dt = 0.01*ms
eqs_leak = leak_current(gl=60*nS, El=10.
↪6*mV, current_name='I_leak')

eqs_K = K_current_HH(gmax=7.2*uS, EK=-
↪12*mV, current_name='I_K')

eqs_Na = Na_current_HH(gmax=24*uS,␣
↪ENa=115*mV, current_name='I_Na')

eqs = (MembraneEquation(C=200*pF) +
       eqs_leak + eqs_K + eqs+Na +
       Current('I_inj : amp'))
``` | ```python
defaultclock.dt = 0.01*ms
gl = 60*nS; El = 10.6*mV
eqs_leak = Equations('I_leak = gl*(El -␣
↪vm) : amp')
g_K = 7.2*uS; EK = -12*mV
eqs_K = Equations('''I_K = g_K*n**4*(EK-
↪vm) : amp

                   dn/dt = alphan*(1-
↪n)-betan*n : 1
                   alphan = .01*(10*mV-
↪vm)/(exp(1-.1*vm/mV)-1)/mV/ms : Hz
                   betan = .125*exp(-.
↪0125*vm/mV)/ms : Hz''')
g_Na = 24*uS; ENa = 115*mV
eqs_Na = Equations('''I_Na = g_
↪Na*m**3*h*(ENa-vm) : amp
                   dm/dt=alpham*(1-m)-
↪betam*m : 1
                   dh/dt=alphah*(1-h)-
↪betah*h : 1
                   alpham=.1*(25*mV-
↪vm)/(exp(2.5-.1*vm/mV)-1)/mV/ms : Hz
                   betam=4*exp(-.
↪0556*vm/mV)/ms : Hz
                   alphah=.07*exp(-.
↪05*vm/mV)/ms : Hz
                   betah=1./(1+exp(3.-
↪.1*vm/mV))/ms : Hz''')
C = 200*pF
eqs = Equations('''dvm/dt = (I_leak + I_
↪K + I_Na + I_inj)/C : volt
                   I_inj : amp''') + eqs_
↪leak + eqs_K + eqs_Na
``` |

Synapses

Brian 1's synaptic models, provided in `brian.library.synpases` can be converted to the equivalent Brian 2 equations as follows:

Current-based synapses

Brian 1	Brian 2

<pre>syn_eqs = exp_current('s', tau=5*ms,
↪current_name='I_syn')
eqs = (MembraneEquation(C=1*nF) +
↪Current('Im = gl*(El-vm) : amp') +
        syn_eqs)
group = NeuronGroup(N, eqs, threshold=
↪'vm>-50*mV', reset='vm=-70*mV')
syn = Synapses(source, group, pre='s +=
↪1*nA')
# ... connect synapses, etc.</pre> 	
	<pre>tau = 5*ms
	syn_eqs = Equations('dI_syn/dt = -I_syn/
	↪tau : amp')
	eqs = (Equations('dvm/dt = (gl*(El - vm)
	↪+ I_syn)/C : volt') +
	        syn_eqs)
	group = NeuronGroup(N, eqs, threshold=
	↪'vm>-50*mV', reset='vm=-70*mV')
	syn = Synapses(source, group, on_pre='I_
	↪syn += 1*nA')
	# ... connect synapses, etc.</pre>

<pre>syn_eqs = alpha_current('s', tau=2.5*ms,
↪current_name='I_syn')
eqs = ... # remaining code as above</pre>
	<pre>tau = 2.5*ms
	syn_eqs = Equations('''dI_syn/dt = (s -
	↪I_syn)/tau : amp
	                        ds/dt = -s/tau :
	↪amp''')
	group = NeuronGroup(N, eqs, threshold=
	↪'vm>-50*mV', reset='vm=-70*mV')
	syn = Synapses(source, group, on_pre='s
	↪+= 1*nA')
	# ... connect synapses, etc.</pre>

<pre>syn_eqs = biexp_current('s', tau1=2.5*ms,
↪ tau2=10*ms, current_name='I_syn')
eqs = ... # remaining code as above</pre>
	<pre>tau1 = 2.5*ms; tau2 = 10*ms; invpeak =
	↪(tau2 / tau1) ** (tau1 / (tau2 - tau1))
	syn_eqs = Equations('''dI_syn/dt =
	↪(invpeak*s - I_syn)/tau1 : amp
	                        ds/dt = -s/tau2 :
	↪amp''')
	eqs = ... # remaining code as above</pre>

Conductance-based synapses

Brian 1	Brian 2

<pre>syn_eqs = exp_conductance('s', tau=5*ms,
↪E=0*mV, conductance_name='g_syn')
eqs = (MembraneEquation(C=1*nF) +
↪Current('Im = gl*(El-vm) : amp') +
        syn_eqs)
group = NeuronGroup(N, eqs, threshold=
↪'vm>-50*mV', reset='vm=-70*mV')
syn = Synapses(source, group, pre='s +=
↪10*nS')
# ... connect synapses, etc.</pre> 	
	<pre>tau = 5*ms; E = 0*mV
	syn_eqs = Equations('dg_syn/dt = -g_syn/
	↪tau : siemens')
	eqs = (Equations('dvm/dt = (gl*(El - vm)
	↪+ g_syn*(E - vm))/C : volt') +
	        syn_eqs)
	group = NeuronGroup(N, eqs, threshold=
	↪'vm>-50*mV', reset='vm=-70*mV')
	syn = Synapses(source, group, on_pre='g_
	↪syn += 10*nS')
	# ... connect synapses, etc.</pre>

Table 4 – continued from previous page

Brian 1	Brian 2

`syn_eqs = alpha_conductance('s', tau=2.`
`↪5*ms, E=0*mV, conductance_name='g_syn')`
`eqs = ... # remaining code as above`
	`tau = 2.5*ms; E = 0*mV`
	`syn_eqs = Equations('''dg_syn/dt = (s -␣`
	`↪g_syn)/tau : siemens`
	`                        ds/dt = -s/tau :␣`
	`↪siemens''')`
	`group = NeuronGroup(N, eqs, threshold=`
	`↪'vm>-50*mV', reset='vm=-70*mV')`
	`syn = Synapses(source, group, on_pre='s␣`
	`↪+= 10*nS')`
	`# ... connect synapses, etc.`

`syn_eqs = biexp_conductance('s', tau1=2.`
`↪5*ms, tau2=10*ms, E=0*mV,`
`                            conductance_`
`↪name='g_syn')`
`eqs = ... # remaining code as above`
	`tau1 = 2.5*ms; tau2 = 10*ms; E = 0*mV`
	`invpeak = (tau2 / tau1) ** (tau1 / (tau2␣`
	`↪- tau1))`
	`syn_eqs = Equations('''dg_syn/dt =␣`
	`↪(invpeak*s - g_syn)/tau1 : siemens`
	`                        ds/dt = -s/tau2 :␣`
	`↪siemens''')`
	`eqs = ... # remaining code as above`

Brian Hears

Deprecated since version 2.2.2.2: Use the brian2hears package instead.

This module is designed for users of the Brian 1 library "Brian Hears". It allows you to use Brian Hears with Brian 2 with only a few modifications (although it's not compatible with the "standalone" mode of Brian 2). The way it works is by acting as a "bridge" to the version in Brian 1. To make this work, you must have a copy of Brian 1 installed (preferably the latest version), and import Brian Hears using:

```python
from brian2.hears import *
```

Many scripts will run without any changes, but there are a few caveats to be aware of. Mostly, the problems are due to the fact that the units system in Brian 2 is not 100% compatible with the units system of Brian 1.

FilterbankGroup now follows the rules for *NeuronGroup* in Brian 2, which means some changes may be necessary to match the syntax of Brian 2, for example, the following would work in Brian 1 Hears:

```python
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset=0, threshold=1, refractory=5*ms)
```

However, in Brian 2 Hears you would need to do:

```python
# Leaky integrate-and-fire model with noise and refractoriness
eqs = '''
dv/dt = (I-v)/(1*ms)+0.2*xi*(2/(1*ms))**.5 : 1 (unless refractory)
I : 1
'''
anf = FilterbankGroup(ihc, 'I', eqs, reset='v=0', threshold='v>1', refractory=5*ms)
```

Slicing sounds no longer works. Previously you could do, e.g. `sound[:20*ms]` but with Brian 2 you would need to do `sound.slice(0*ms, 20*ms)`.

In addition, some functions may not work correctly with Brian 2 units. In most circumstances, Brian 2 units can be used interchangeably with Brian 1 units in the bridge, but in some cases it may be necessary to convert units from one format to another, and to do that you can use the functions *convert_unit_b1_to_b2* and *convert_unit_b2_to_b1*.

1.5 Known issues

In addition to the issues noted below, you can refer to our bug tracker on GitHub.

> **List of known issues**
>
> - *Cannot find msvcr90d.dll*
> - *"AttributeError: MSVCCompiler instance has no attribute 'compiler_cxx'"*
> - *"Missing compiler_cxx fix for MSVCCompiler"*
> - *Problems with numerical integration*
> - *Jupyter notebooks and C++ standalone mode progress reporting*
> - *Parallel Brian simulations with C++ standalone*
> - *Parallel Brian simulations with Cython on machines with NFS (e.g. a computing cluster)*
> - *Slow C++ standalone simulations*
> - *Cython fails with compilation error on OS X:* `error: use of undeclared identifier 'isinf'`
> - *CMD windows open when running Brian on Windows with the Spyder 3 IDE*

1.5.1 Cannot find msvcr90d.dll

If you see this message coming up, find the file `PythonDir\Lib\site-packages\numpy\distutils\mingw32ccompiler.py` and modify the line `msvcr_dbg_success = build_msvcr_library(debug=True)` to read `msvcr_dbg_success = False` (you can comment out the existing line and add the new line immediately after).

1.5.2 "AttributeError: MSVCCompiler instance has no attribute 'compiler_cxx'"

This is caused by a bug in some versions of numpy on Windows. The easiest solution is to update to the latest version of numpy.

If that isn't possible, a hacky solution is to modify the numpy code directly to fix the problem. The following change may work. Modify line 388 of `numpy/distutils/ccompiler.py` from `elif not self.compiler_cxx:` to `elif not hasattr(self, 'compiler_cxx') or not self.compiler_cxx:`. If the line number is different, it should be nearby. Search for `elif not self.compiler_cxx` in that file.

1.5.3 "Missing compiler_cxx fix for MSVCCompiler"

If you keep seeing this message, do not worry. It's not possible for us to hide it, but doesn't indicate any problems.

1.5.4 Problems with numerical integration

In some cases, the automatic choice of numerical integration method will not be appropriate, because of a choice of parameters that couldn't be determined in advance. In this case, typically you will get nan (not a number) values in the results, or large oscillations. In this case, Brian will generate a warning to let you know, but will not raise an error.

1.5.5 Jupyter notebooks and C++ standalone mode progress reporting

When you run simulations in C++ standalone mode and enable progress reporting (e.g. by using `report='text'` as a keyword argument), the progress will not be displayed in the jupyter notebook. If you started the notebook from a terminal, you will find the output there. Unfortunately, this is a tricky problem to solve at the moment, due to the details of how the jupyter notebook handles output.

1.5.6 Parallel Brian simulations with C++ standalone

Simulations using the C++ standalone device will create code and store results in a dedicated directory (`output`, by default). If you run multiple simulations in parallel, you have to take care that these simulations do not use the same directory – otherwise, everything from compilation errors to incorrect results can happen. Either chose a different directory name for each simulation and provide it as the `directory` argument to the *set_device* or *build* call, or use `directory=None` which will use a randomly chosen unique temporary directory (in `/tmp` on Unix-based systems) for each simulation. If you need to know the directory name, you can access it after the simulation run via `device.project_dir`.

1.5.7 Parallel Brian simulations with Cython on machines with NFS (e.g. a computing cluster)

Generated Cython code is stored in a cache directory on disk so that it can be reused when it is needed again, without recompiling it. Multiple simulations running in parallel could interfere during the compilation process by trying to generate the same file at the same time. To avoid this, Brian uses a file locking mechanism that ensures that only a process at a time can access these files. Unfortunately, this file locking mechanism is very slow on machines using the Network File System (NFS), which is often the case on computing clusters. On such machines, it is recommend to use an independent cache directory per process, and to disable the file locking mechanism. This can be done with the following code that has to be run at the beginning of each process:

```
from brian2 import *
import os
cache_dir = os.path.expanduser(f'~/.cython/brian-pid-{os.getpid()}')
prefs.codegen.runtime.cython.cache_dir = cache_dir
prefs.codegen.runtime.cython.multiprocess_safe = False
```

1.5.8 Slow C++ standalone simulations

Some versions of the GNU standard library (in particular those used by recent Ubuntu versions) have a bug that can dramatically slow down simulations in C++ standalone mode on modern hardware (see #803). As a workaround, Brian will set an environment variable `LD_BIND_NOW` during the execution of standalone simulations which changes the way the library is linked so that it does not suffer from this problem. If this environment variable leads to unwanted behaviour on your machine, change the `prefs.devices.cpp_standalone.run_environment_variables` preference.

1.5.9 Cython fails with compilation error on OS X: `error: use of undeclared identifier 'isinf'`

Try setting the environment variable `MACOSX_DEPLOYMENT_TARGET=10.9`.

1.5.10 CMD windows open when running Brian on Windows with the Spyder 3 IDE

This is due to the interaction with the integrated ipython terminal. Either change the run configuration to "Execute in an external system terminal" or patch the internal Python function used to spawn processes as described in github issue #1140.

1.6 Support

If you are stuck with a problem using Brian, please do get in touch at our community forum.

You can save time by following this procedure when reporting a problem:

1. Do try to solve the problem on your own first. Read the documentation, including using the search feature, index and reference documentation.

2. Search the mailing list archives to see if someone else already had the same problem.

3. Before writing, try to create a minimal example that reproduces the problem. You'll get the fastest response if you can send just a handful of lines of code that show what isn't working.

1.6.1 Which version of Brian am I using?

When reporting problems, it is important to include the information what exact version of Brian you are using. The different install methods listed in *Installation* provide different mechanisms to get this information. For example, if you used `conda` for installing Brian, you can use `conda list brian2`; if you used `pip`, you can use `pip show brian2`.

A general method that works independent of the installation method is to ask the Brian package itself:

```
>>> import brian2
>>> print(brian2.__version__)
2.4.2
```

This method also has the advantage that you can easily call it from the same environment (e.g. an IDE or a Jupyter Notebook) that you use when you execute Brian scripts. This helps avoiding mistakes where you think you use a specific version but in fact you use a different one. In such cases, it can also be helpful to look at Brian's `__file__` attribute:

```
>>> print(brian2.__file__)
/home/marcel/anaconda3/envs/brian2_test/lib/python3.9/site-packages/brian2/__init__.py
```

In the above example, it shows that the `brian2` installation in the conda environment `brian2_test` is used.

If you installed a *development version* of Brian, then the version number will contain additional information:

```
>>> print(brian2.__version__)
2.4.2.post0.dev408
```

The above means that the Brian version that is used has 408 additional commits that were added after the 2.4.2 release. To get the exact git commit for the local Brian installation, use:

```
>>> print(brian2.__git_revision__)
d2cb4a85f804037ef055503975d822ff3f473ccf
```

To get more information about this commit, you can append it to the repository URL on GitHub as `/commit/<commit id>` (where the first few characters of the `<commit id>` are enough), e.g. for the commit referenced above: https://github.com/brian-team/brian2/commit/d2cb4a85

1.7 Compatibility and reproducibility

1.7.1 Supported Python and numpy versions

We follow the approach outlined in numpy's deprecation policy. This means that Brian supports:

- All minor versions of Python released 42 months prior to Brian, and at minimum the two latest minor versions.

- All minor versions of numpy released in the 24 months prior to Brian, and at minimum the last three minor versions.

Note that we do not have control about the versions that are supported by the conda-forge infrastructure. Therefore, `brian2` conda packages might not be provided for all of the supported versions. In this case, affected users can chose to either update the Python/numpy version in their conda environment to a version with a conda package or to install `brian2` via pip.

1.7.2 General policy

We try to keep backwards-incompatible changes to a minimum. In general, `brian2` scripts should continue to work with newer versions and should give the same results.

As an exception to the above rule, we will always correct clearly identified bugs that lead to incorrect simulation results (i.e., not just an matter of interpretation). Since we do not want to require new users to take any action to get correct results, we will change the default behaviour in such cases. If possible, we will give the user an option to restore the old, incorrect behaviour to reproduce the previous results with newer Brian versions. This would typically be a preference in the `legacy` category, see *legacy.refractory_timing* for an example.

Note: The order of terms when evaluating equations is not fixed and can change with the version of `sympy`, the symbolic mathematics library used in Brian. Similarly, Brian performs a number of optimizations by default and asks the compiler to perform further ones which might introduce subtle changes depending on the compiler and its version. Finally, code generation can lead to either Python or C++ code (with a single or multiple threads) executing the actual simulation which again may affect the numerical results. Therefore, we cannot guarantee exact, "bitwise" reproducibility of results.

1.7.3 Syntax deprecations

We sometimes realize that the names of arguments or other syntax elements are confusing and therefore decide to change them. In such cases, we start to use the new syntax everywhere in the documentation and examples, but leave the former syntax available for compatiblity with previously written code. For example, earlier versions of Brian used `method='linear'` to describe the exact solution of differential equations via sympy (that most importantly applies to "linear" equations, i.e. linear differential equations with constant coefficients). However, some users interpreted `method='linear'` as a "linear approximation" like the forward Euler method. In newer versions of Brian the recommended syntax is therefore to use `method='exact'`, but the old syntax remains valid.

If the changed syntax is very prominent, its continued use in Brian scripts (published by others) could be confusing to new users. In these cases, we might decide to give a warning when the deprecated syntax is used (e.g. for the `pre` and `post` arguments in *Synapses* which have been replaced by `on_pre` and `on_post`). Such warnings will contain all the information necessary to rewrite the code so that the warning is no longer raised (in line with our general *policy for warnings*).

1.7.4 Random numbers

Streams of random numbers in Brian simulations (including the generation of synapses, etc.) are reproducible when a seed is set via Brian's *seed()* function. Note that there is a difference with regard to random numbers between *runtime and standalone mode*: in runtime mode, numpy's random number generator is always used – even from generated Cython code. Therefore, the call to *seed()* will set numpy's random number generator seed which then applies to all random numbers. Regardless of whether initial values of a variable are set via an explicit call to `numpy.random.randn`, or via a Brian expression such as `'randn()'`, both are affected by this seed. In contrast, random numbers in standalone simulations will be generated by an independent random number generator (but based on the same algorithm as numpy's) and the call to *seed()* will only affect these numbers, not numbers resulting from explicit calls to `numpy.random`. To make standalone scripts mixing both sources of randomness reproducible, either set numpy's random generator seed manually in addition to calling *seed()*, or reformulate the model to use code generation everywhere (e.g. replace `group.v = -70*mV + 10*mV*np.random.randn(len(group))` by `group.v = '-70*mv + 10*mV*randn()'`).

Changing the code generation target can imply a change in the order in which random numbers are drawn from the reproducible random number stream. In general, we therefore only guarantee the use of the same numbers if the code generation target and the number of threads (for C++ standalone simulations) is the same.

Note: If there are several sources of randomness (e.g. multiple *PoissonGroup* objects) in a simulation, then the order in which these elements are executed matters. The order of execution is deterministic, but if it is not unambiguously determined by the `when` and `order` attributes (see *Scheduling* for details), then it will depend on the names of objects. When not explicitly given via the `name` argument during the object's creation, names are automatically generated by Brian as e.g. `poissongroup`, `poissongroup_1`, etc. When you repeatedly run simulations within the same process, these names might change and therefore the order in which the elements are simulated. Random numbers will then be differently distributed to the objects. To avoid this and get reproducible random number streams you can either fix the order of elements by specifying the `order` or `name` argument, or make sure that each simulation gets run in a fresh Python process.

1.7.5 Python errors

While we try to guarantee the reproducibility of simulations (within the limits stated above), we do so only for code that does not raise any error. We constantly try to improve the error handling in Brian, and these improvements can lead to errors raised at a different time (e.g. when creating an object as opposed to when running the simulation), different types of errors being raised (e.g. `DimensionMismatchError` instead of `TypeError`), or simply a different error message text. Therefore, Brian scripts should never use `try`/`except` blocks to implement program logic.

1.8 Contributor Covenant Code of Conduct

1.8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

1.8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

1.8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

1.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

1.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at team@briansimulator.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

1.8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

TUTORIALS

The tutorial consists of a series of Jupyter Notebooks[1].

For more information about how to use Jupyter Notebooks, see the Jupyter Notebook documentation.

2.1 Introduction to Brian part 1: Neurons

All Brian scripts start with the following. If you're trying this notebook out in the Jupyter notebook, you should start by running this cell.

```python
from brian2 import *
```

Later we'll do some plotting in the notebook, so we activate inline plotting in the notebook by doing this:

```python
%matplotlib inline
```

If you are not using the Jupyter notebook to run this example (e.g. you are using a standard Python terminal, or you copy&paste these example into an editor and run them as a script), then plots will not automatically be displayed. In this case, call the show() command explicitly after the plotting commands.

2.1.1 Units system

Brian has a system for using quantities with physical dimensions:

```python
20*volt
```

$$20.0\,\mathrm{V}$$

All of the basic SI units can be used (volt, amp, etc.) along with all the standard prefixes (m=milli, p=pico, etc.), as well as a few special abbreviations like mV for millivolt, pF for picofarad, etc.

```python
1000*amp
```

$$1.0\,\mathrm{k\,A}$$

```python
1e6*volt
```

$$1.0\,\mathrm{M\,V}$$

---

[1] Formerly known as "IPython Notebooks".

```
1000*namp
```

$$1.0000000000000002\,\mu\,\text{A}$$

Also note that combinations of units with work as expected:

```
10*nA*5*Mohm
```

$$49.99999999999999\,\text{m V}$$

And if you try to do something wrong like adding amps and volts, what happens?

```
5*amp+10*volt
```

```
---------------------------------------------------------------------------

DimensionMismatchError                    Traceback (most recent call last)

<ipython-input-8-245c0c0332d1> in <module>
----> 1 5*amp+10*volt


~/programming/brian2/brian2/units/fundamentalunits.py in __add__(self, other)
   1429
   1430     def __add__(self, other):
-> 1431         return self._binary_operation(other, operator.add,
   1432                                       fail_for_mismatch=True,
   1433                                       operator_str='+')


~/programming/brian2/brian2/units/fundamentalunits.py in _binary_operation(self,␣
↪other, operation, dim_operation, fail_for_mismatch, operator_str, inplace)
   1369             message = ('Cannot calculate {value1} %s {value2}, units do␣
↪not '
   1370                       'match') % operator_str
-> 1371             _, other_dim = fail_for_dimension_mismatch(self, other,␣
↪message,
   1372                                                        value1=self,
   1373                                                        value2=other)


~/programming/brian2/brian2/units/fundamentalunits.py in fail_for_dimension_
↪mismatch(obj1, obj2, error_message, **error_quantities)
    184             raise DimensionMismatchError(error_message, dim1)
    185         else:
--> 186             raise DimensionMismatchError(error_message, dim1, dim2)
    187     else:
    188         return dim1, dim2


DimensionMismatchError: Cannot calculate 5. A + 10. V, units do not match (units are␣
↪A and V).
```

If you haven't see an error message in Python before that can look a bit overwhelming, but it's actually quite simple and it's important to know how to read these because you'll probably see them quite often.

You should start at the bottom and work up. The last line gives the error type `DimensionMismatchError` along with a more specific message (in this case, you were trying to add together two quantities with different SI units, which is impossible).

Working upwards, each of the sections starts with a filename (e.g. `C:\Users\Dan\...`) with possibly the name of a function, and then a few lines surrounding the line where the error occurred (which is identified with an arrow).

The last of these sections shows the place in the function where the error actually happened. The section above it shows the function that called that function, and so on until the first section will be the script that you actually run. This sequence of sections is called a traceback, and is helpful in debugging.

If you see a traceback, what you want to do is start at the bottom and scan up the sections until you find your own file because that's most likely where the problem is. (Of course, your code might be correct and Brian may have a bug in which case, please let us know on the email support list.)

2.1.2 A simple model

Let's start by defining a simple neuron model. In Brian, all models are defined by systems of differential equations. Here's a simple example of what that looks like:

```
tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''
```

In Python, the notation `'''` is used to begin and end a multi-line string. So the equations are just a string with one line per equation. The equations are formatted with standard mathematical notation, with one addition. At the end of a line you write `:  unit` where `unit` is the SI unit of that variable. Note that this is not the unit of the two sides of the equation (which would be $1/second$), but the unit of the *variable* defined by the equation, i.e. in this case v.

Now let's use this definition to create a neuron.

```
G = NeuronGroup(1, eqs)
```

In Brian, you only create groups of neurons, using the class `NeuronGroup`. The first two arguments when you create one of these objects are the number of neurons (in this case, 1) and the defining differential equations.

Let's see what happens if we didn't put the variable `tau` in the equation:

```
eqs = '''
dv/dt = 1-v : 1
'''
G = NeuronGroup(1, eqs)
run(100*ms)
```

```
---------------------------------------------------------------------

DimensionMismatchError                    Traceback (most recent call last)

~/programming/brian2/brian2/equations/equations.py in check_units(self, group, run_
↪namespace)
    955                 try:
--> 956                     check_dimensions(str(eq.expr), self.dimensions[var] /␣
↪second.dim,
    957                                      all_variables)


~/programming/brian2/brian2/equations/unitcheck.py in check_dimensions(expression,␣
↪dimensions, variables)
     44                                      expected=repr(get_
↪unit(dimensions)))
```

```
---> 45         fail_for_dimension_mismatch(expr_dims, dimensions, err_msg)
     46


~/programming/brian2/brian2/units/fundamentalunits.py in fail_for_dimension_
→mismatch(obj1, obj2, error_message, **error_quantities)
    183         if obj2 is None or isinstance(obj2, (Dimension, Unit)):
--> 184             raise DimensionMismatchError(error_message, dim1)
    185         else:


DimensionMismatchError: Expression 1-v does not have the expected unit hertz (unit is␣
→1).


During handling of the above exception, another exception occurred:


DimensionMismatchError                    Traceback (most recent call last)
~/programming/brian2/brian2/core/network.py in before_run(self, run_namespace)
    897                 try:
--> 898                     obj.before_run(run_namespace)
    899                 except Exception as ex:


~/programming/brian2/brian2/groups/neurongroup.py in before_run(self, run_namespace)
    883         # Check units
--> 884         self.equations.check_units(self, run_namespace=run_namespace)
    885         # Check that subexpressions that refer to stateful functions are␣
→labeled


~/programming/brian2/brian2/equations/equations.py in check_units(self, group, run_
→namespace)
    958             except DimensionMismatchError as ex:
--> 959                 raise DimensionMismatchError(('Inconsistent units in '
    960                                               'differential equation '


DimensionMismatchError: Inconsistent units in differential equation defining variable␣
→v:
Expression 1-v does not have the expected unit hertz (unit is 1).


During handling of the above exception, another exception occurred:


BrianObjectException                      Traceback (most recent call last)
<ipython-input-11-97ed109f5888> in <module>
      3 '''
      4 G = NeuronGroup(1, eqs)
----> 5 run(100*ms)


~/programming/brian2/brian2/units/fundamentalunits.py in new_f(*args, **kwds)
```

```
  2383                                                        get_
↪dimensions(newkeyset[k]))
  2384
-> 2385              result = f(*args, **kwds)
  2386              if 'result' in au:
  2387                  if au['result'] == bool:


~/programming/brian2/brian2/core/magic.py in run(duration, report, report_period,␣
↪namespace, profile, level)
   371          intended use. See `MagicNetwork` for more details.
   372      '''
--> 373      return magic_network.run(duration, report=report, report_period=report_
↪period,
   374                               namespace=namespace, profile=profile,␣
↪level=2+level)
   375 run.__module__ = __name__


~/programming/brian2/brian2/core/magic.py in run(self, duration, report, report_
↪period, namespace, profile, level)
   229              namespace=None, profile=False, level=0):
   230          self._update_magic_objects(level=level+1)
--> 231          Network.run(self, duration, report=report, report_period=report_
↪period,
   232                      namespace=namespace, profile=profile, level=level+1)
   233


~/programming/brian2/brian2/core/base.py in device_override_decorated_function(*args,␣
↪**kwds)
   274                  return getattr(curdev, name)(*args, **kwds)
   275              else:
--> 276                  return func(*args, **kwds)
   277
   278          device_override_decorated_function.__doc__ = func.__doc__


~/programming/brian2/brian2/units/fundamentalunits.py in new_f(*args, **kwds)
  2383                                                        get_
↪dimensions(newkeyset[k]))
  2384
-> 2385              result = f(*args, **kwds)
  2386              if 'result' in au:
  2387                  if au['result'] == bool:


~/programming/brian2/brian2/core/network.py in run(self, duration, report, report_
↪period, namespace, profile, level)
  1007              namespace = get_local_namespace(level=level+3)
  1008
-> 1009          self.before_run(namespace)
  1010
  1011          if len(all_objects) == 0:


~/programming/brian2/brian2/core/base.py in device_override_decorated_function(*args,␣
↪**kwds)
```

```
    274                    return getattr(curdev, name)(*args, **kwds)
    275                else:
--> 276                    return func(*args, **kwds)
    277
    278           device_override_decorated_function.__doc__ = func.__doc__


~/programming/brian2/brian2/core/network.py in before_run(self, run_namespace)
    898                    obj.before_run(run_namespace)
    899                except Exception as ex:
--> 900                    raise brian_object_exception("An error occurred when␣
→preparing an object.", obj, ex)
    901
    902           # Check that no object has been run as part of another network before


BrianObjectException: Original error and traceback:
Traceback (most recent call last):
  File "/home/marcel/programming/brian2/brian2/equations/equations.py", line 956, in␣
→check_units
    check_dimensions(str(eq.expr), self.dimensions[var] / second.dim,
  File "/home/marcel/programming/brian2/brian2/equations/unitcheck.py", line 45, in␣
→check_dimensions
    fail_for_dimension_mismatch(expr_dims, dimensions, err_msg)
  File "/home/marcel/programming/brian2/brian2/units/fundamentalunits.py", line 184,␣
→in fail_for_dimension_mismatch
    raise DimensionMismatchError(error_message, dim1)
brian2.units.fundamentalunits.DimensionMismatchError: Expression 1-v does not have␣
→the expected unit hertz (unit is 1).


During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/marcel/programming/brian2/brian2/core/network.py", line 898, in before_
→run
    obj.before_run(run_namespace)
  File "/home/marcel/programming/brian2/brian2/groups/neurongroup.py", line 884, in␣
→before_run
    self.equations.check_units(self, run_namespace=run_namespace)
  File "/home/marcel/programming/brian2/brian2/equations/equations.py", line 959, in␣
→check_units
    raise DimensionMismatchError(('Inconsistent units in '
brian2.units.fundamentalunits.DimensionMismatchError: Inconsistent units in␣
→differential equation defining variable v:
Expression 1-v does not have the expected unit hertz (unit is 1).


Error encountered with object named "neurongroup_1".
Object was created here (most recent call only, full details in debug log):
  File "<ipython-input-11-97ed109f5888>", line 4, in <module>
    G = NeuronGroup(1, eqs)


An error occurred when preparing an object. brian2.units.fundamentalunits.
→DimensionMismatchError: Inconsistent units in differential equation defining␣
→variable v:
Expression 1-v does not have the expected unit hertz (unit is 1).
(See above for original error message and traceback.)
```

An error is raised, but why? The reason is that the differential equation is now dimensionally inconsistent. The left hand side `dv/dt` has units of `1/second` but the right hand side `1-v` is dimensionless. People often find this behaviour of Brian confusing because this sort of equation is very common in mathematics. However, for quantities with physical dimensions it is incorrect because the results would change depending on the unit you measured it in. For time, if you measured it in seconds the same equation would behave differently to how it would if you measured time in milliseconds. To avoid this, we insist that you always specify dimensionally consistent equations.

Now let's go back to the good equations and actually run the simulation.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs)
run(100*ms)
```

```
INFO        No numerical integration method specified for group 'neurongroup', using␣
→method 'exact' (took 0.02s). [brian2.stateupdaters.base.method_choice]
```

First off, ignore that `start_scope()` at the top of the cell. You'll see that in each cell in this tutorial where we run a simulation. All it does is make sure that any Brian objects created before the function is called aren't included in the next run of the simulation.

Secondly, you'll see that there is an "INFO" message about not specifying the numerical integration method. This is harmless and just to let you know what method we chose, but we'll fix it in the next cell by specifying the method explicitly.

So, what has happened here? Well, the command `run(100*ms)` runs the simulation for 100 ms. We can see that this has worked by printing the value of the variable `v` before and after the simulation.

```
start_scope()

G = NeuronGroup(1, eqs, method='exact')
print('Before v = %s' % G.v[0])
run(100*ms)
print('After v = %s' % G.v[0])
```

```
Before v = 0.0
After v = 0.9999546000702376
```

By default, all variables start with the value 0. Since the differential equation is `dv/dt=(1-v)/tau` we would expect after a while that `v` would tend towards the value 1, which is just what we see. Specifically, we'd expect `v` to have the value `1-exp(-t/tau)`. Let's see if that's right.

```
print('Expected value of v = %s' % (1-exp(-100*ms/tau)))
```

```
Expected value of v = 0.9999546000702375
```

Good news, the simulation gives the value we'd expect!

Now let's take a look at a graph of how the variable `v` evolves over time.

```
start_scope()
```

```
G = NeuronGroup(1, eqs, method='exact')
M = StateMonitor(G, 'v', record=True)

run(30*ms)

plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v');
```
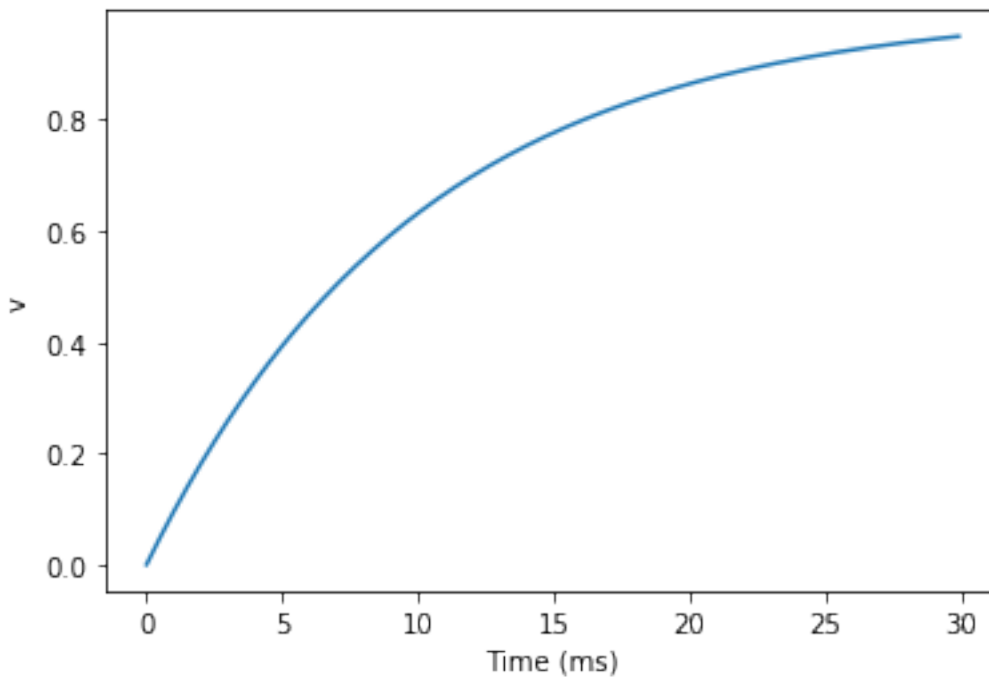

This time we only ran the simulation for 30 ms so that we can see the behaviour better. It looks like it's behaving as expected, but let's just check that analytically by plotting the expected behaviour on top.

```
start_scope()

G = NeuronGroup(1, eqs, method='exact')
M = StateMonitor(G, 'v', record=0)

run(30*ms)

plot(M.t/ms, M.v[0], 'C0', label='Brian')
plot(M.t/ms, 1-exp(-M.t/tau), 'C1--',label='Analytic')
xlabel('Time (ms)')
ylabel('v')
legend();
```
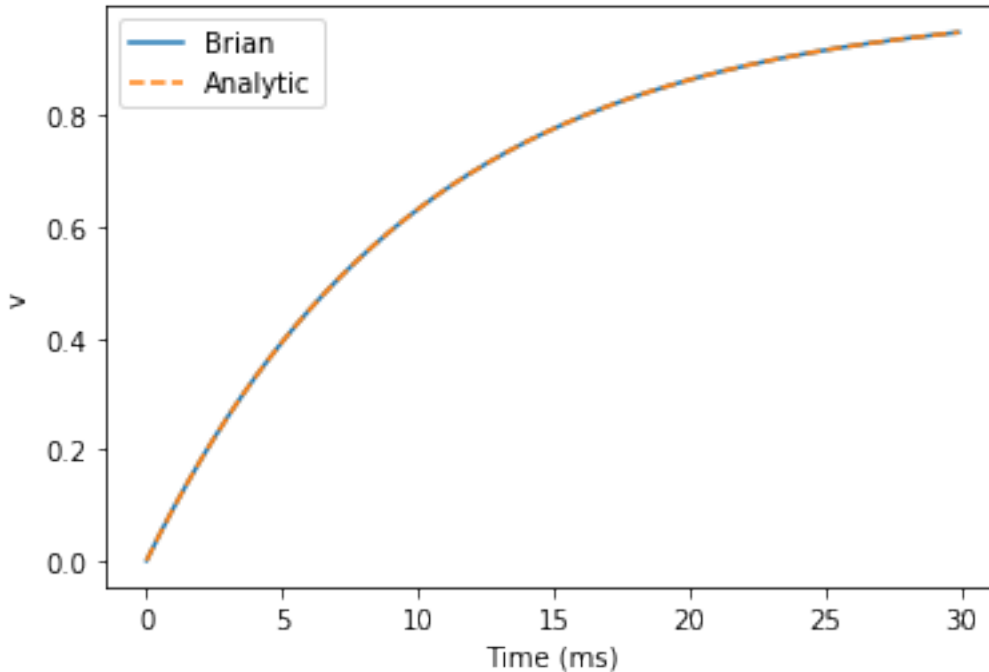
As you can see, the blue (Brian) and dashed orange (analytic solution) lines coincide.

In this example, we used the object `StateMonitor` object. This is used to record the values of a neuron variable while the simulation runs. The first two arguments are the group to record from, and the variable you want to record from. We also specify `record=0`. This means that we record all values for neuron 0. We have to specify which neurons we want to record because in large simulations with many neurons it usually uses up too much RAM to record the values of all neurons.

Now try modifying the equations and parameters and see what happens in the cell below.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (sin(2*pi*100*Hz*t)-v)/tau : 1
'''

# Change to Euler method because exact integrator doesn't work here
G = NeuronGroup(1, eqs, method='euler')
M = StateMonitor(G, 'v', record=0)

G.v = 5 # initial value

run(60*ms)

plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v');
```
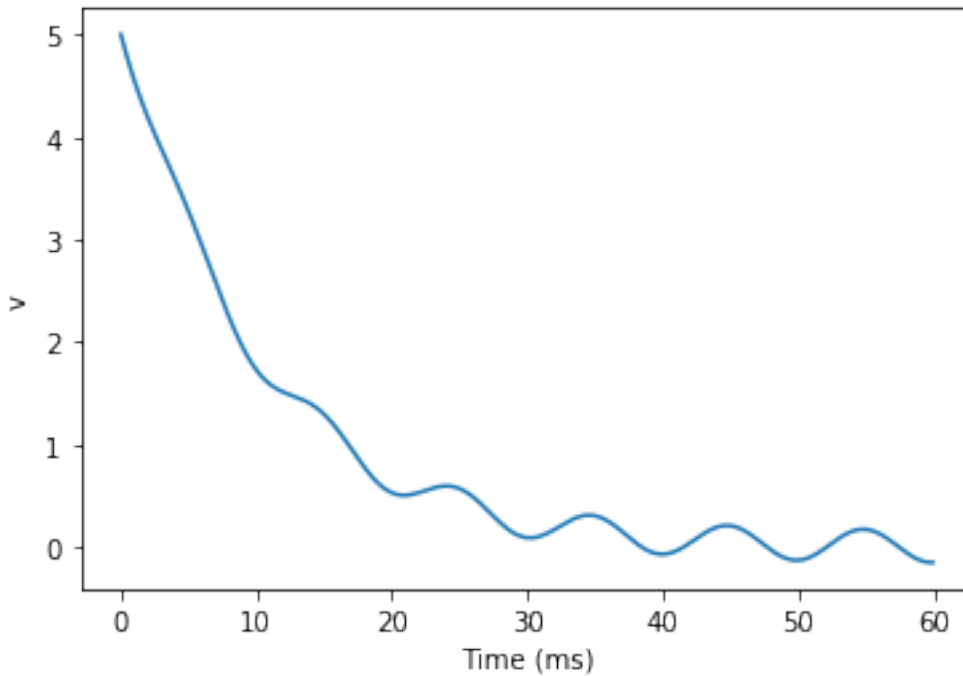
2.1.3 Adding spikes

So far we haven't done anything neuronal, just played around with differential equations. Now let's start adding spiking behaviour.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', method='exact')

M = StateMonitor(G, 'v', record=0)
run(50*ms)
plot(M.t/ms, M.v[0])
xlabel('Time (ms)')
ylabel('v');
```
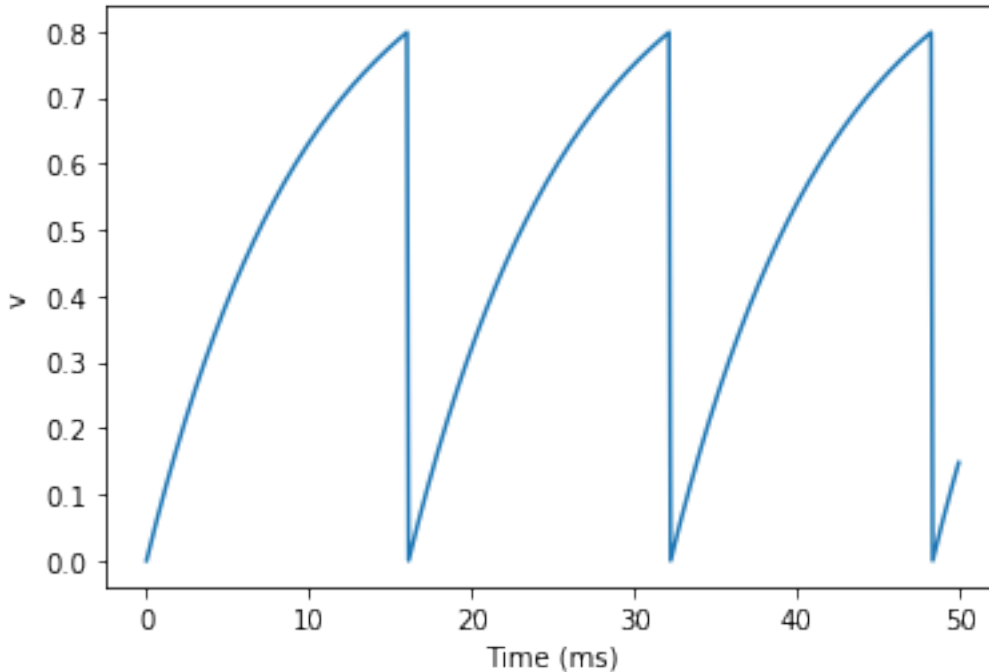
We've added two new keywords to the `NeuronGroup` declaration: `threshold='v>0.8'` and `reset='v = 0'`. What this means is that when `v>0.8` we fire a spike, and immediately reset `v = 0` after the spike. We can put any expression and series of statements as these strings.

As you can see, at the beginning the behaviour is the same as before until `v` crosses the threshold `v>0.8` at which point you see it reset to 0. You can't see it in this figure, but internally Brian has registered this event as a spike. Let's have a look at that.

```
start_scope()

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', method='exact')

spikemon = SpikeMonitor(G)

run(50*ms)

print('Spike times: %s' % spikemon.t[:])
```

```
Spike times: [16.  32.1 48.2] ms
```

The `SpikeMonitor` object takes the group whose spikes you want to record as its argument and stores the spike times in the variable `t`. Let's plot those spikes on top of the other figure to see that it's getting it right.

```
start_scope()

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', method='exact')

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
```

```
for t in spikemon.t:
    axvline(t/ms, ls='--', c='C1', lw=3)
xlabel('Time (ms)')
ylabel('v');
```
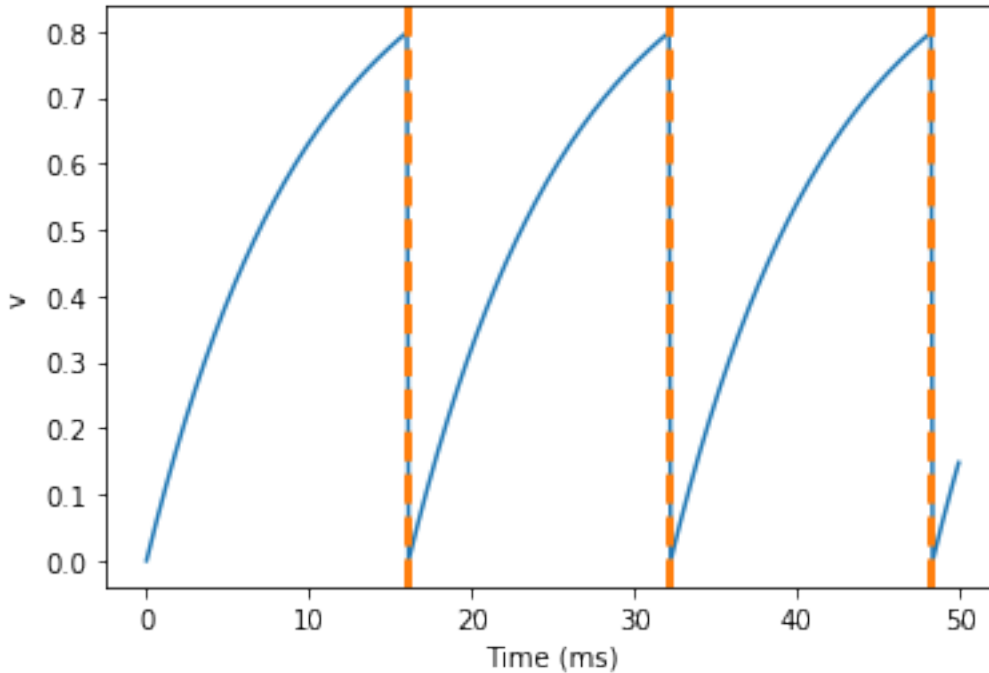

Here we've used the `axvline` command from `matplotlib` to draw an orange, dashed vertical line at the time of each spike recorded by the `SpikeMonitor`.

Now try changing the strings for `threshold` and `reset` in the cell above to see what happens.

2.1.4 Refractoriness

A common feature of neuron models is refractoriness. This means that after the neuron fires a spike it becomes refractory for a certain duration and cannot fire another spike until this period is over. Here's how we do that in Brian.

```
start_scope()

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1 (unless refractory)
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', refractory=5*ms, method=
↪'exact')

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
```

```
for t in spikemon.t:
    axvline(t/ms, ls='--', c='C1', lw=3)
xlabel('Time (ms)')
ylabel('v');
```
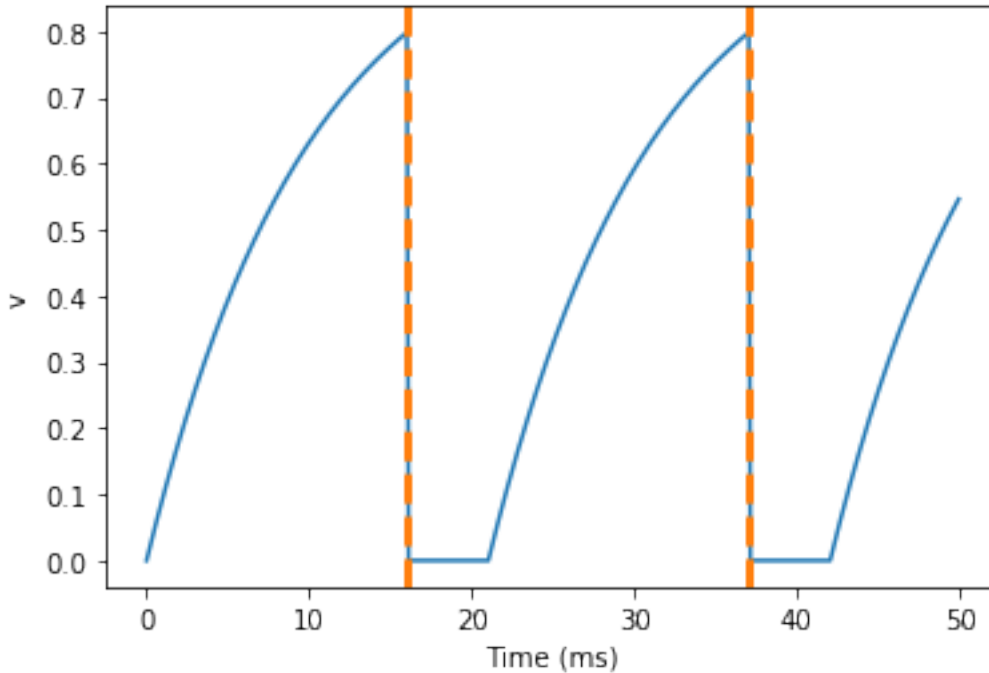

As you can see in this figure, after the first spike, v stays at 0 for around 5 ms before it resumes its normal behaviour. To do this, we've done two things. Firstly, we've added the keyword `refractory=5*ms` to the `NeuronGroup` declaration. On its own, this only means that the neuron cannot spike in this period (see below), but doesn't change how v behaves. In order to make v stay constant during the refractory period, we have to add `(unless refractory)` to the end of the definition of v in the differential equations. What this means is that the differential equation determines the behaviour of v unless it's refractory in which case it is switched off.

Here's what would happen if we didn't include `(unless refractory)`. Note that we've also decreased the value of `tau` and increased the length of the refractory period to make the behaviour clearer.

```
start_scope()

tau = 5*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''

G = NeuronGroup(1, eqs, threshold='v>0.8', reset='v = 0', refractory=15*ms, method=
→'exact')

statemon = StateMonitor(G, 'v', record=0)
spikemon = SpikeMonitor(G)

run(50*ms)

plot(statemon.t/ms, statemon.v[0])
for t in spikemon.t:
```

2.1. Introduction to Brian part 1: Neurons

```
    axvline(t/ms, ls='--', c='C1', lw=3)
axhline(0.8, ls=':', c='C2', lw=3)
xlabel('Time (ms)')
ylabel('v')
print("Spike times: %s" % spikemon.t[:])
```

```
Spike times: [ 8. 23. 38.] ms
```
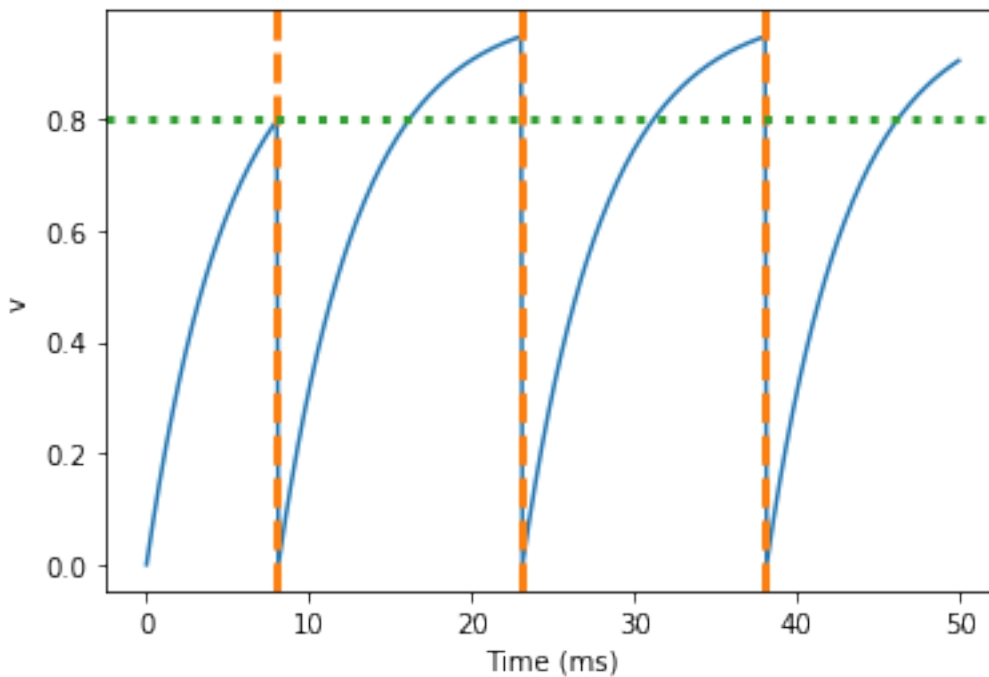

So what's going on here? The behaviour for the first spike is the same: v rises to 0.8 and then the neuron fires a spike at time 8 ms before immediately resetting to 0. Since the refractory period is now 15 ms this means that the neuron won't be able to spike again until time 8 + 15 = 23 ms. Immediately after the first spike, the value of v now instantly starts to rise because we didn't specify (unless refractory) in the definition of dv/dt. However, once it reaches the value 0.8 (the dashed green line) at time roughly 8 ms it doesn't fire a spike even though the threshold is v>0.8. This is because the neuron is still refractory until time 23 ms, at which point it fires a spike.

Note that you can do more complicated and interesting things with refractoriness. See the full documentation for more details about how it works.

2.1.5 Multiple neurons

So far we've only been working with a single neuron. Let's do something interesting with multiple neurons.

```
start_scope()

N = 100
tau = 10*ms
eqs = '''
dv/dt = (2-v)/tau : 1
'''
```

```
G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', method='exact')
G.v = 'rand()'

spikemon = SpikeMonitor(G)

run(50*ms)

plot(spikemon.t/ms, spikemon.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index');
```
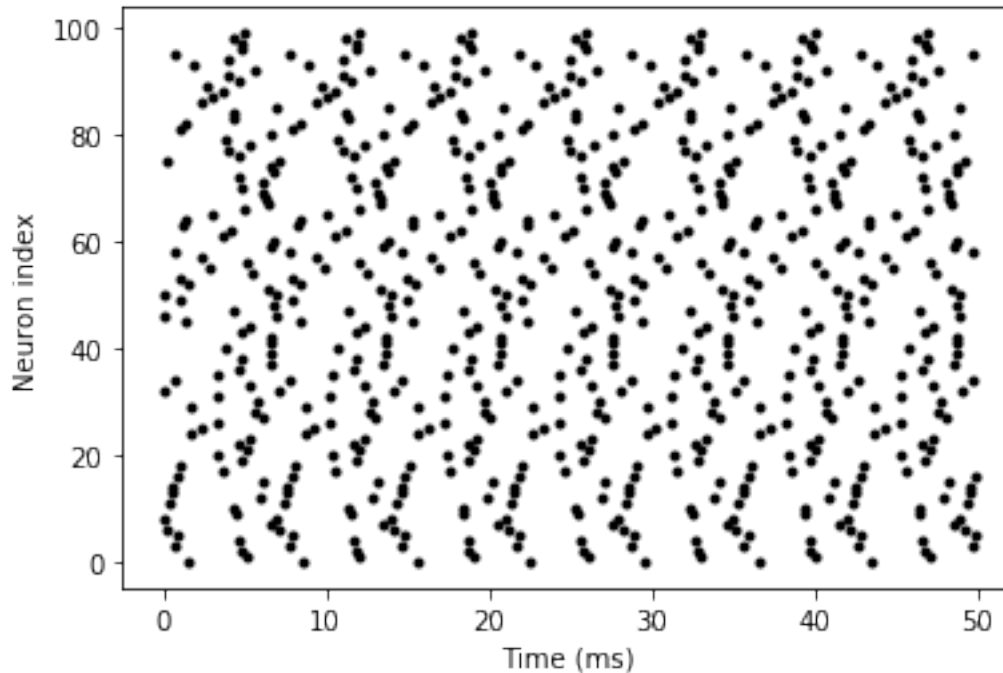

This shows a few changes. Firstly, we've got a new variable `N` determining the number of neurons. Secondly, we added the statement `G.v = 'rand()'` before the run. What this does is initialise each neuron with a different uniform random value between 0 and 1. We've done this just so each neuron will do something a bit different. The other big change is how we plot the data in the end.

As well as the variable `spikemon.t` with the times of all the spikes, we've also used the variable `spikemon.i` which gives the corresponding neuron index for each spike, and plotted a single black dot with time on the x-axis and neuron index on the y-value. This is the standard "raster plot" used in neuroscience.

2.1.6 Parameters

To make these multiple neurons do something more interesting, let's introduce per-neuron parameters that don't have a differential equation attached to them.

```
start_scope()

N = 100
tau = 10*ms
v0_max = 3.
```

```
duration = 1000*ms

eqs = '''
dv/dt = (v0-v)/tau : 1 (unless refractory)
v0 : 1
'''

G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', refractory=5*ms, method='exact')
M = SpikeMonitor(G)

G.v0 = 'i*v0_max/(N-1)'

run(duration)

figure(figsize=(12,4))
subplot(121)
plot(M.t/ms, M.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(122)
plot(G.v0, M.count/duration)
xlabel('v0')
ylabel('Firing rate (sp/s)');
```
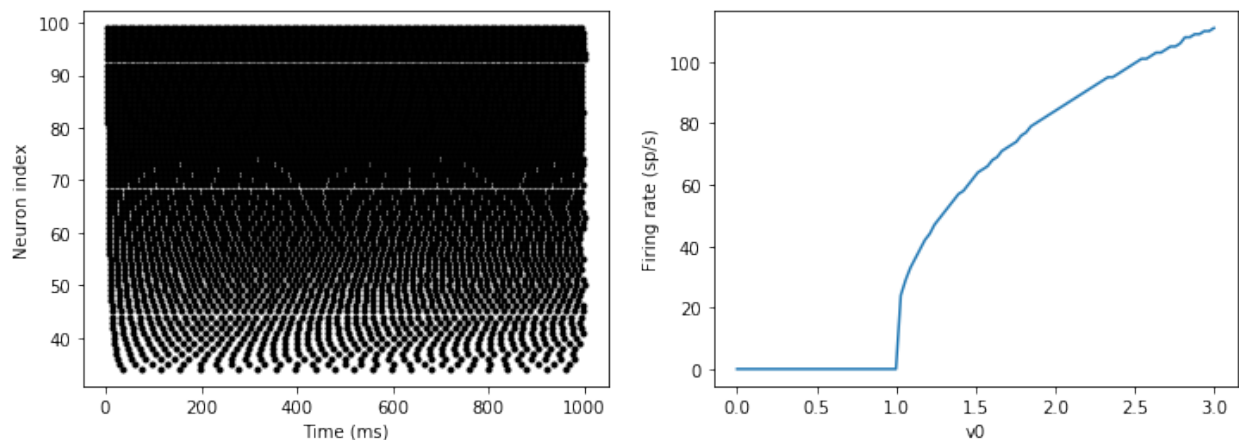

The line `v0 : 1` declares a new per-neuron parameter `v0` with units `1` (i.e. dimensionless).

The line `G.v0 = 'i*v0_max/(N-1)'` initialises the value of v0 for each neuron varying from 0 up to `v0_max`. The symbol `i` when it appears in strings like this refers to the neuron index.

So in this example, we're driving the neuron towards the value `v0` exponentially, but when `v` crosses `v>1`, it fires a spike and resets. The effect is that the rate at which it fires spikes will be related to the value of `v0`. For `v0<1` it will never fire a spike, and as `v0` gets larger it will fire spikes at a higher rate. The right hand plot shows the firing rate as a function of the value of `v0`. This is the I-f curve of this neuron model.

Note that in the plot we've used the `count` variable of the `SpikeMonitor`: this is an array of the number of spikes each neuron in the group fired. Dividing this by the duration of the run gives the firing rate.

2.1.7 Stochastic neurons

Often when making models of neurons, we include a random element to model the effect of various forms of neural noise. In Brian, we can do this by using the symbol `xi` in differential equations. Strictly speaking, this symbol is a "stochastic differential" but you can sort of thinking of it as just a Gaussian random variable with mean 0 and standard deviation 1. We do have to take into account the way stochastic differentials scale with time, which is why we multiply it by `tau**-0.5` in the equations below (see a textbook on stochastic differential equations for more details). Note that we also changed the `method` keyword argument to use `'euler'` (which stands for the Euler-Maruyama method); the `'exact'` method that we used earlier is not applicable to stochastic differential equations.

```
start_scope()

N = 100
tau = 10*ms
v0_max = 3.
duration = 1000*ms
sigma = 0.2

eqs = '''
dv/dt = (v0-v)/tau+sigma*xi*tau**-0.5 : 1 (unless refractory)
v0 : 1
'''

G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', refractory=5*ms, method='euler')
M = SpikeMonitor(G)

G.v0 = 'i*v0_max/(N-1)'

run(duration)

figure(figsize=(12,4))
subplot(121)
plot(M.t/ms, M.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(122)
plot(G.v0, M.count/duration)
xlabel('v0')
ylabel('Firing rate (sp/s)');
```
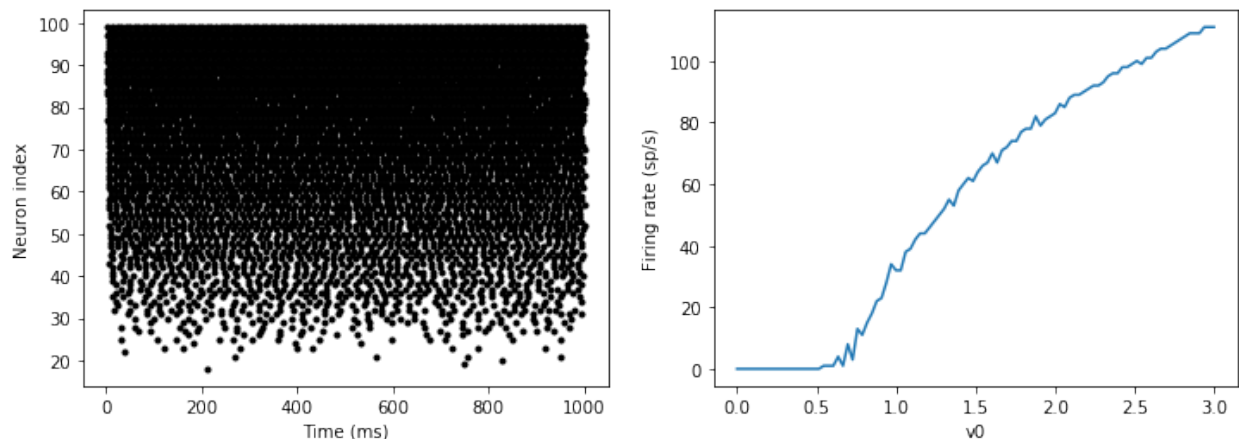

That's the same figure as in the previous section but with some noise added. Note how the curve has changed shape: instead of a sharp jump from firing at rate 0 to firing at a positive rate, it now increases in a sigmoidal fashion. This is

because no matter how small the driving force the randomness may cause it to fire a spike.

2.1.8 End of tutorial

That's the end of this part of the tutorial. The cell below has another example. See if you can work out what it is doing and why. Try adding a `StateMonitor` to record the values of the variables for one of the neurons to help you understand it.

You could also try out the things you've learned in this cell.

Once you're done with that you can move on to the next tutorial on Synapses.

```python
start_scope()

N = 1000
tau = 10*ms
vr = -70*mV
vt0 = -50*mV
delta_vt0 = 5*mV
tau_t = 100*ms
sigma = 0.5*(vt0-vr)
v_drive = 2*(vt0-vr)
duration = 100*ms

eqs = '''
dv/dt = (v_drive+vr-v)/tau + sigma*xi*tau**-0.5 : volt
dvt/dt = (vt0-vt)/tau_t : volt
'''

reset = '''
v = vr
vt += delta_vt0
'''

G = NeuronGroup(N, eqs, threshold='v>vt', reset=reset, refractory=5*ms, method='euler
↪')
spikemon = SpikeMonitor(G)

G.v = 'rand()*(vt0-vr)+vr'
G.vt = vt0

run(duration)

_ = hist(spikemon.t/ms, 100, histtype='stepfilled', facecolor='k',␣
↪weights=list(ones(len(spikemon))/(N*defaultclock.dt)))
xlabel('Time (ms)')
ylabel('Instantaneous firing rate (sp/s)');
```
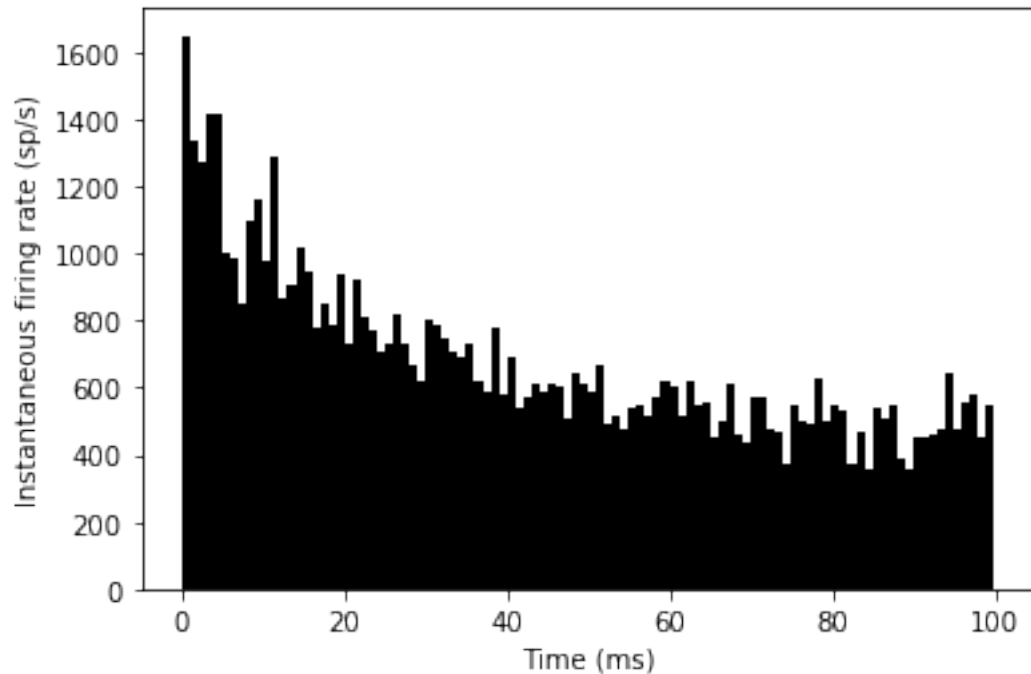
2.2 Introduction to Brian part 2: Synapses

If you haven't yet read part 1: Neurons, go read that now.

As before we start by importing the Brian package and setting up matplotlib for IPython:

```
from brian2 import *
%matplotlib inline
```

2.2.1 The simplest Synapse

Once you have some neurons, the next step is to connect them up via synapses. We'll start out with doing the simplest possible type of synapse that causes an instantaneous change in a variable after a spike.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(2, eqs, threshold='v>1', reset='v = 0', method='exact')
G.I = [2, 0]
G.tau = [10, 100]*ms

# Comment these two lines out to see what happens without Synapses
S = Synapses(G, G, on_pre='v_post += 0.2')
S.connect(i=0, j=1)
```

```
M = StateMonitor(G, 'v', record=True)

run(100*ms)

plot(M.t/ms, M.v[0], label='Neuron 0')
plot(M.t/ms, M.v[1], label='Neuron 1')
xlabel('Time (ms)')
ylabel('v')
legend();
```

```
<matplotlib.legend.Legend at 0x7fdccb8773d0>
```
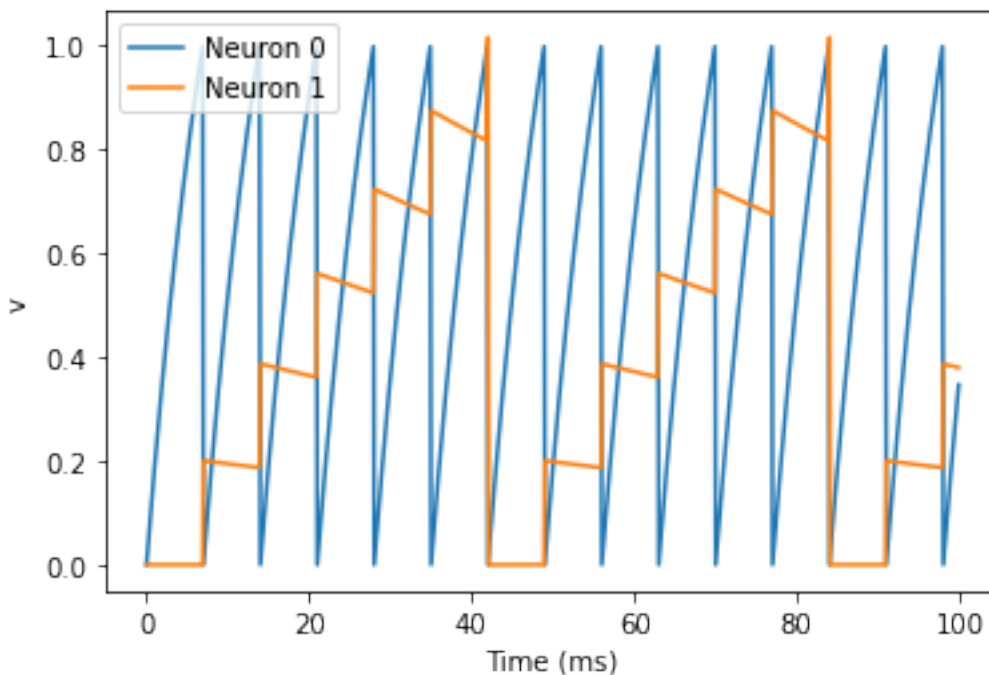

There are a few things going on here. First of all, let's recap what is going on with the `NeuronGroup`. We've created two neurons, each of which has the same differential equation but different values for parameters I and tau. Neuron 0 has I=2 and `tau=10*ms` which means that is driven to repeatedly spike at a fairly high rate. Neuron 1 has I=0 and `tau=100*ms` which means that on its own - without the synapses - it won't spike at all (the driving current I is 0). You can prove this to yourself by commenting out the two lines that define the synapse.

Next we define the synapses: `Synapses(source, target, ...)` means that we are defining a synaptic model that goes from `source` to `target`. In this case, the source and target are both the same, the group G. The syntax `on_pre='v_post += 0.2'` means that when a spike occurs in the presynaptic neuron (hence `on_pre`) it causes an instantaneous change to happen `v_post += 0.2`. The `_post` means that the value of v referred to is the post-synaptic value, and it is increased by 0.2. So in total, what this model says is that whenever two neurons in G are connected by a synapse, when the source neuron fires a spike the target neuron will have its value of v increased by 0.2.

However, at this point we have only defined the synapse model, we haven't actually created any synapses. The next line `S.connect(i=0, j=1)` creates a synapse from neuron 0 to neuron 1.

2.2.2 Adding a weight

In the previous section, we hard coded the weight of the synapse to be the value 0.2, but often we would to allow this to be different for different synapses. We do that by introducing synapse equations.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(3, eqs, threshold='v>1', reset='v = 0', method='exact')
G.I = [2, 0, 0]
G.tau = [10, 100, 100]*ms

# Comment these two lines out to see what happens without Synapses
S = Synapses(G, G, 'w : 1', on_pre='v_post += w')
S.connect(i=0, j=[1, 2])
S.w = 'j*0.2'

M = StateMonitor(G, 'v', record=True)

run(50*ms)

plot(M.t/ms, M.v[0], label='Neuron 0')
plot(M.t/ms, M.v[1], label='Neuron 1')
plot(M.t/ms, M.v[2], label='Neuron 2')
xlabel('Time (ms)')
ylabel('v')
legend();
```

```
<matplotlib.legend.Legend at 0x7fdccb7f2750>
```
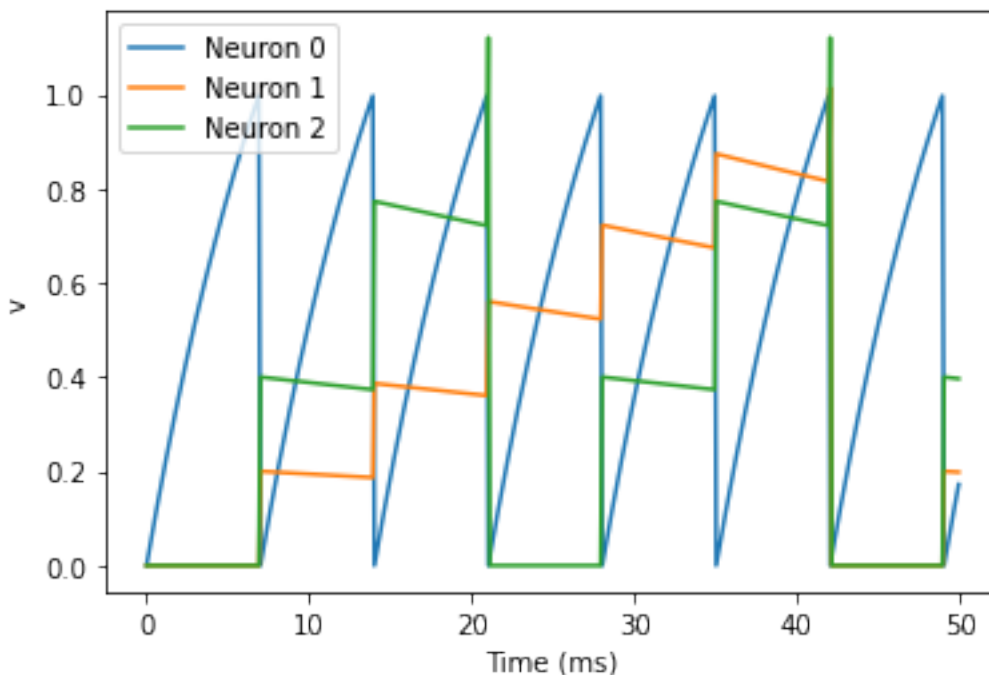
This example behaves very similarly to the previous example, but now there's a synaptic weight variable `w`. The string `'w : 1'` is an equation string, precisely the same as for neurons, that defines a single dimensionless parameter `w`. We changed the behaviour on a spike to `on_pre='v_post += w'` now, so that each synapse can behave differently depending on the value of `w`. To illustrate this, we've made a third neuron which behaves precisely the same as the second neuron, and connected neuron 0 to both neurons 1 and 2. We've also set the weights via `S.w = 'j*0.2'`. When `i` and `j` occur in the context of synapses, `i` refers to the source neuron index, and `j` to the target neuron index. So this will give a synaptic connection from 0 to 1 with weight `0.2=0.2*1` and from 0 to 2 with weight `0.4=0.2*2`.

2.2.3 Introducing a delay

So far, the synapses have been instantaneous, but we can also make them act with a certain delay.

```
start_scope()

eqs = '''
dv/dt = (I-v)/tau : 1
I : 1
tau : second
'''
G = NeuronGroup(3, eqs, threshold='v>1', reset='v = 0', method='exact')
G.I = [2, 0, 0]
G.tau = [10, 100, 100]*ms

S = Synapses(G, G, 'w : 1', on_pre='v_post += w')
S.connect(i=0, j=[1, 2])
S.w = 'j*0.2'
S.delay = 'j*2*ms'

M = StateMonitor(G, 'v', record=True)

run(50*ms)

plot(M.t/ms, M.v[0], label='Neuron 0')
plot(M.t/ms, M.v[1], label='Neuron 1')
plot(M.t/ms, M.v[2], label='Neuron 2')
xlabel('Time (ms)')
ylabel('v')
legend();
```

```
<matplotlib.legend.Legend at 0x7fdccb7f2290>
```
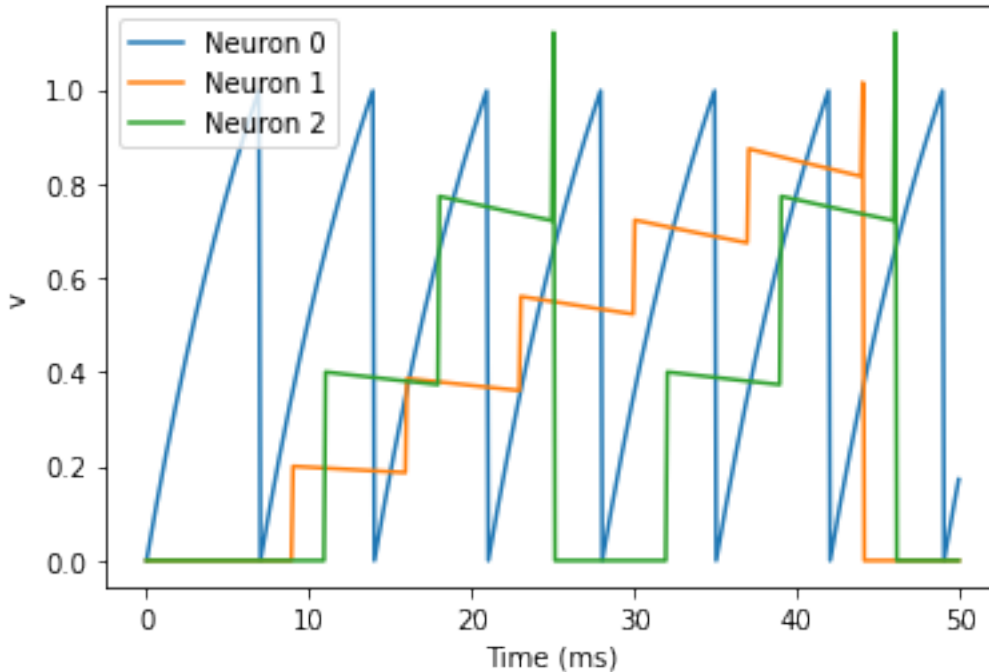
As you can see, that's as simple as adding a line `S.delay = 'j*2*ms'` so that the synapse from 0 to 1 has a delay of 2 ms, and from 0 to 2 has a delay of 4 ms.

2.2.4 More complex connectivity

So far, we specified the synaptic connectivity explicitly, but for larger networks this isn't usually possible. For that, we usually want to specify some condition.

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')
S = Synapses(G, G)
S.connect(condition='i!=j', p=0.2)
```

Here we've created a dummy neuron group of N neurons and a dummy synapses model that doens't actually do anything just to demonstrate the connectivity. The line `S.connect(condition='i!=j', p=0.2)` will connect all pairs of neurons i and j with probability 0.2 as long as the condition `i!=j` holds. So, how can we see that connectivity? Here's a little function that will let us visualise it.

```
def visualise_connectivity(S):
    Ns = len(S.source)
    Nt = len(S.target)
    figure(figsize=(10, 4))
    subplot(121)
    plot(zeros(Ns), arange(Ns), 'ok', ms=10)
    plot(ones(Nt), arange(Nt), 'ok', ms=10)
    for i, j in zip(S.i, S.j):
        plot([0, 1], [i, j], '-k')
    xticks([0, 1], ['Source', 'Target'])
    ylabel('Neuron index')
    xlim(-0.1, 1.1)
```

(continues on next page)

```
    ylim(-1, max(Ns, Nt))
    subplot(122)
    plot(S.i, S.j, 'ok')
    xlim(-1, Ns)
    ylim(-1, Nt)
    xlabel('Source neuron index')
    ylabel('Target neuron index')

visualise_connectivity(S)
```
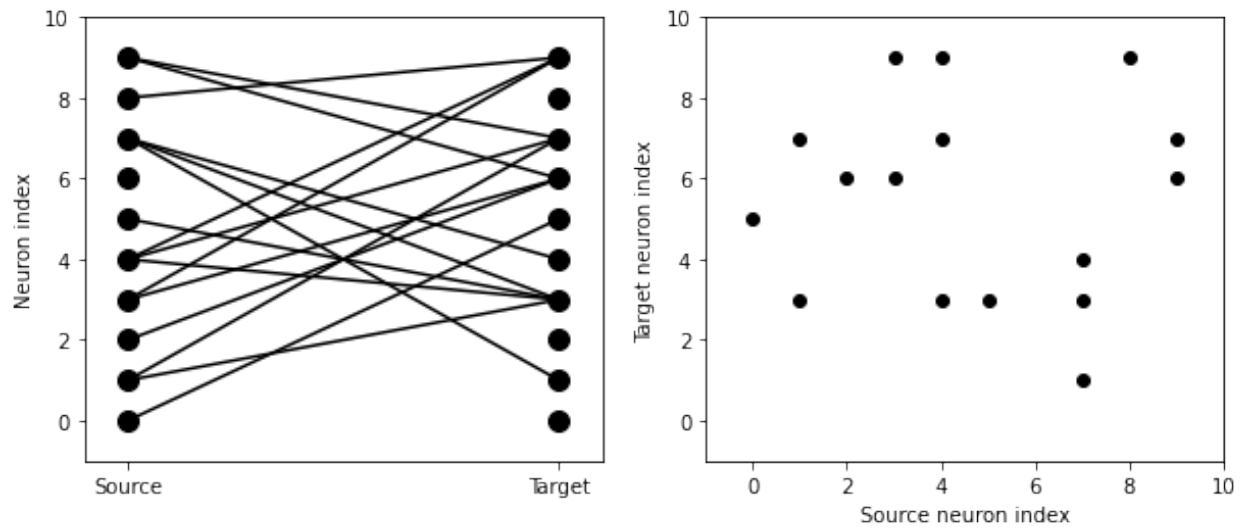

There are two plots here. On the left hand side, you see a vertical line of circles indicating source neurons on the left, and a vertical line indicating target neurons on the right, and a line between two neurons that have a synapse. On the right hand side is another way of visualising the same thing. Here each black dot is a synapse, with x value the source neuron index, and y value the target neuron index.

Let's see how these figures change as we change the probability of a connection:

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

for p in [0.1, 0.5, 1.0]:
    S = Synapses(G, G)
    S.connect(condition='i!=j', p=p)
    visualise_connectivity(S)
    suptitle('p = '+str(p));
```
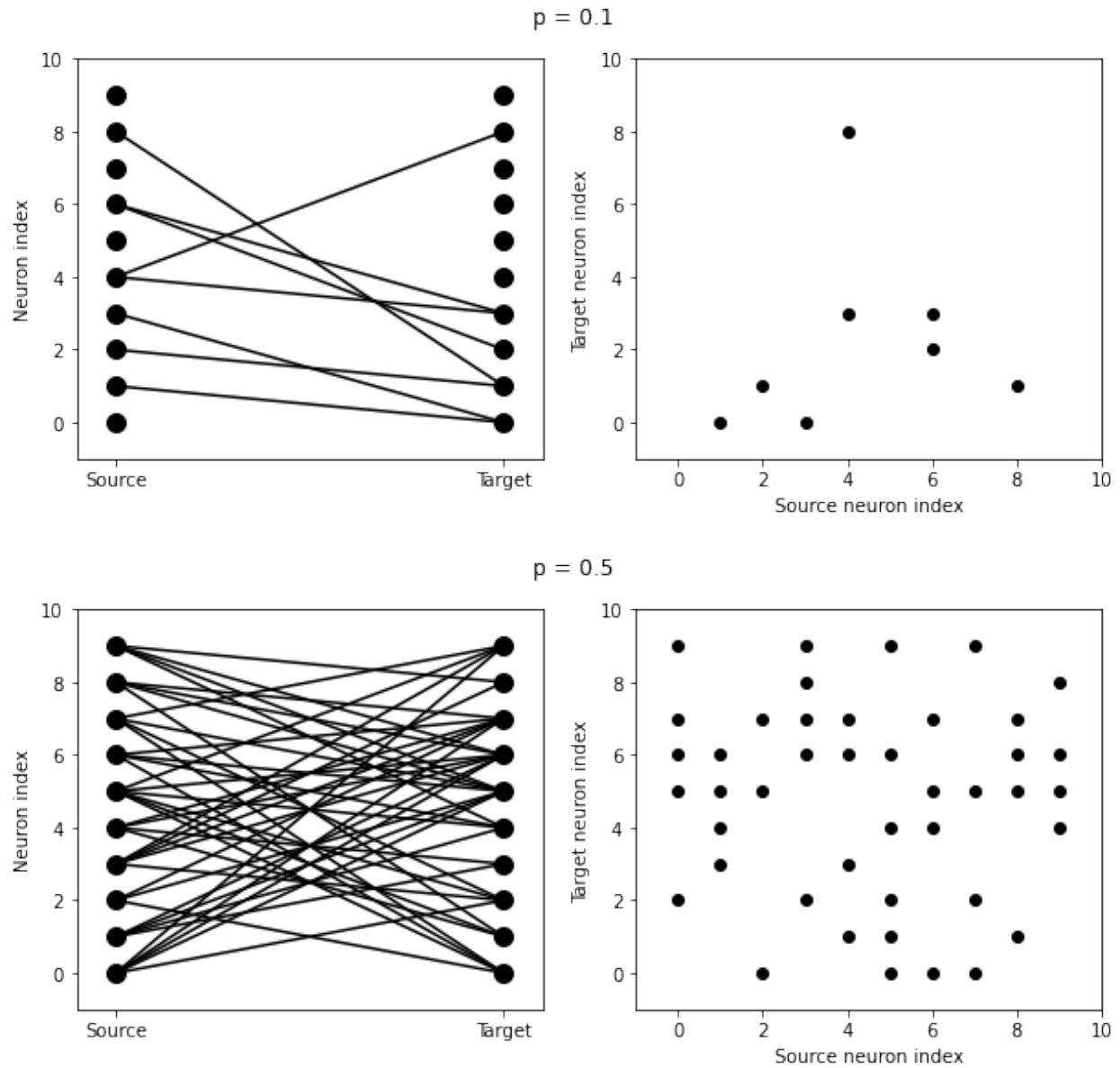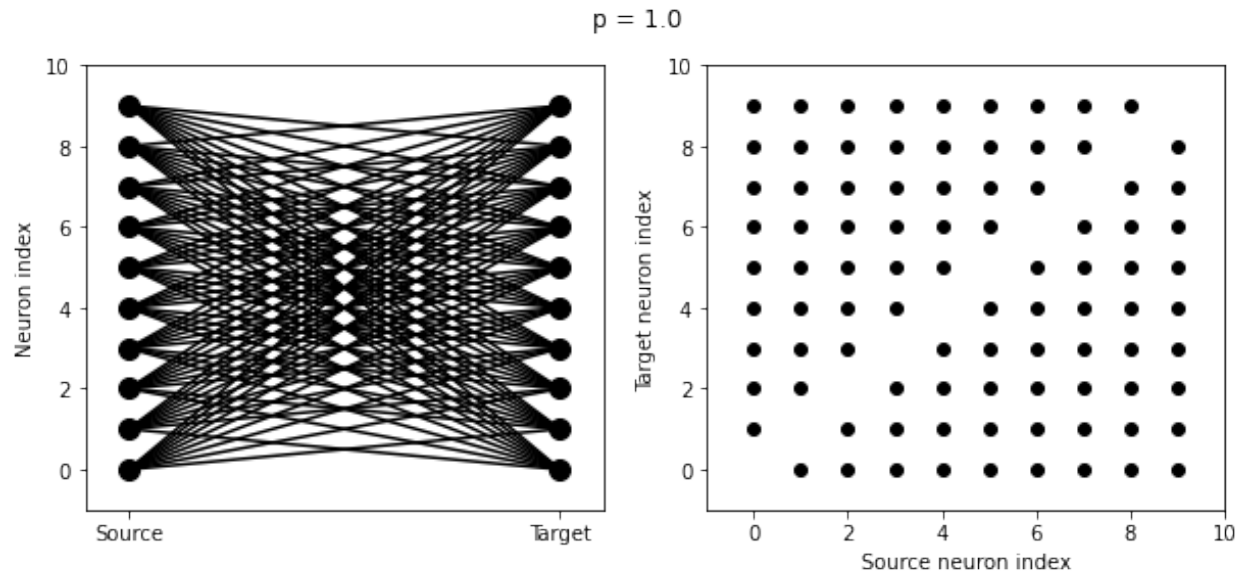
And let's see what another connectivity condition looks like. This one will only connect neighbouring neurons.

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

S = Synapses(G, G)
S.connect(condition='abs(i-j)<4 and i!=j')
visualise_connectivity(S)
```
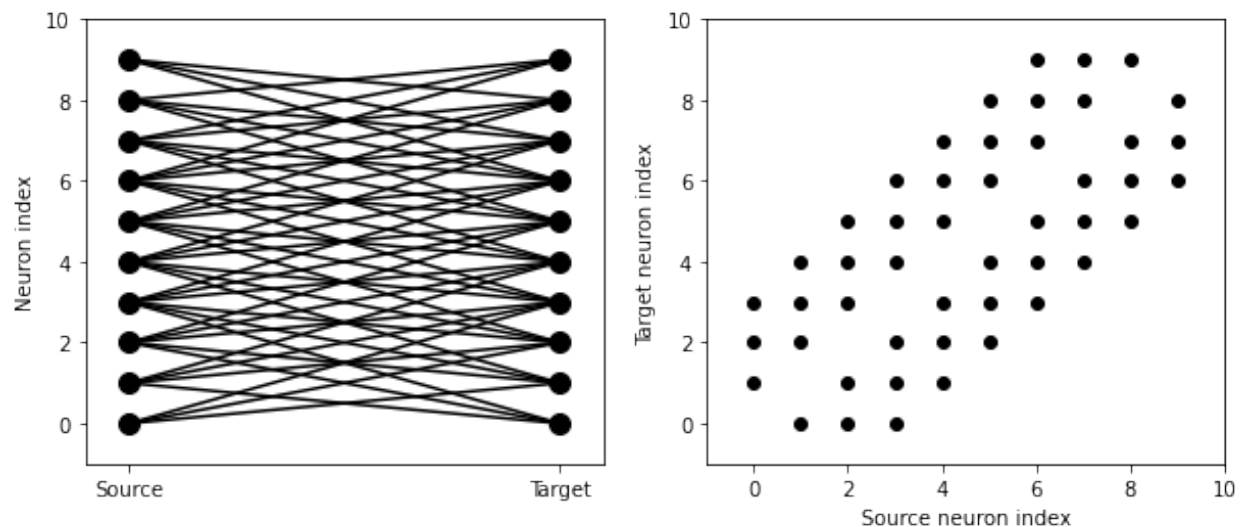

Try using that cell to see how other connectivity conditions look like.

You can also use the generator syntax to create connections like this more efficiently. In small examples like this, it doesn't matter, but for large numbers of neurons it can be much more efficient to specify directly which neurons should be connected than to specify just a condition. Note that the following example uses `skip_if_invalid` to avoid errors at the boundaries (e.g. do not try to connect the neuron with index 1 to a neuron with index -2).

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

S = Synapses(G, G)
S.connect(j='k for k in range(i-3, i+4) if i!=k', skip_if_invalid=True)
visualise_connectivity(S)
```
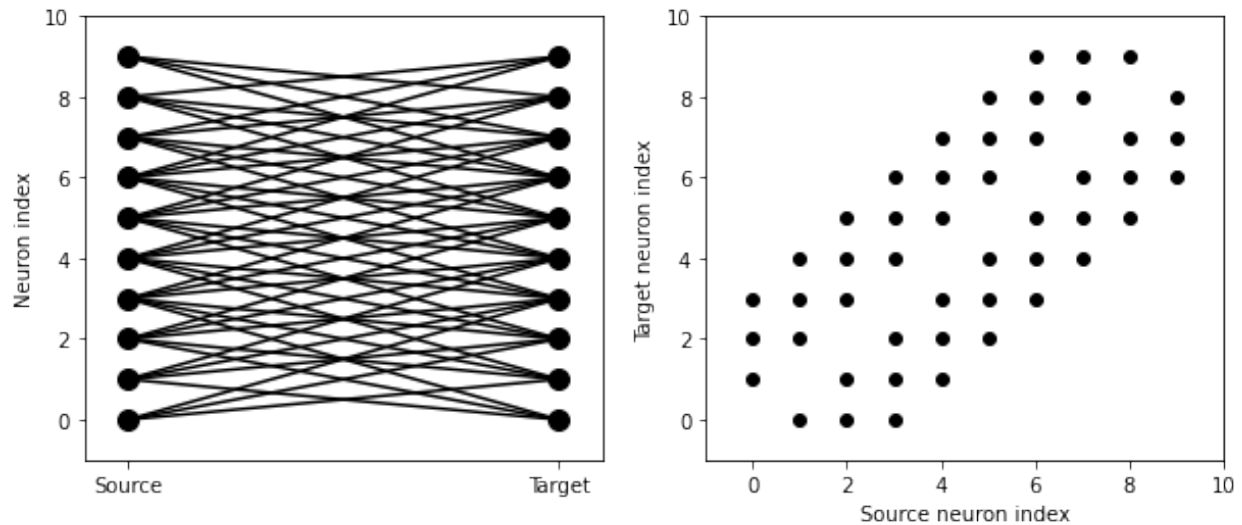

If each source neuron is connected to precisely one target neuron (which would be normally used with two separate groups of the same size, not with identical source and target groups as in this example), there is a special syntax that is extremely efficient. For example, 1-to-1 connectivity looks like this:

```
start_scope()

N = 10
G = NeuronGroup(N, 'v:1')

S = Synapses(G, G)
S.connect(j='i')
visualise_connectivity(S)
```
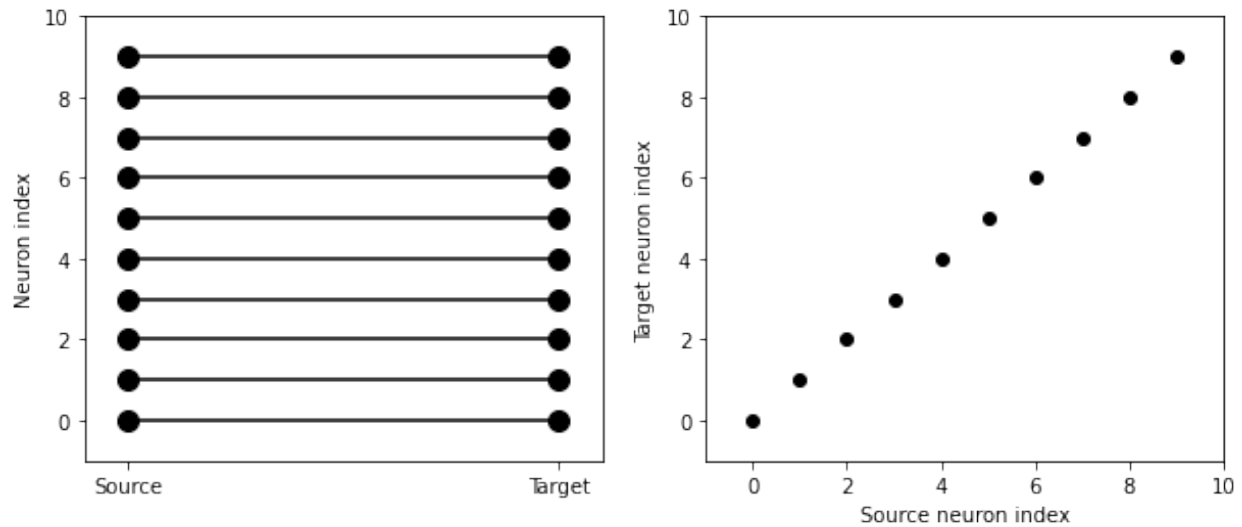
You can also do things like specifying the value of weights with a string. Let's see an example where we assign each neuron a spatial location and have a distance-dependent connectivity function. We visualise the weight of a synapse by the size of the marker.

```
start_scope()

N = 30
neuron_spacing = 50*umetre
width = N/4.0*neuron_spacing

# Neuron has one variable x, its position
G = NeuronGroup(N, 'x : metre')
G.x = 'i*neuron_spacing'

# All synapses are connected (excluding self-connections)
S = Synapses(G, G, 'w : 1')
S.connect(condition='i!=j')
# Weight varies with distance
S.w = 'exp(-(x_pre-x_post)**2/(2*width**2))'

scatter(S.x_pre/um, S.x_post/um, S.w*20)
xlabel('Source neuron position (um)')
ylabel('Target neuron position (um)');
```

```
Text(0, 0.5, 'Target neuron position (um)')
```
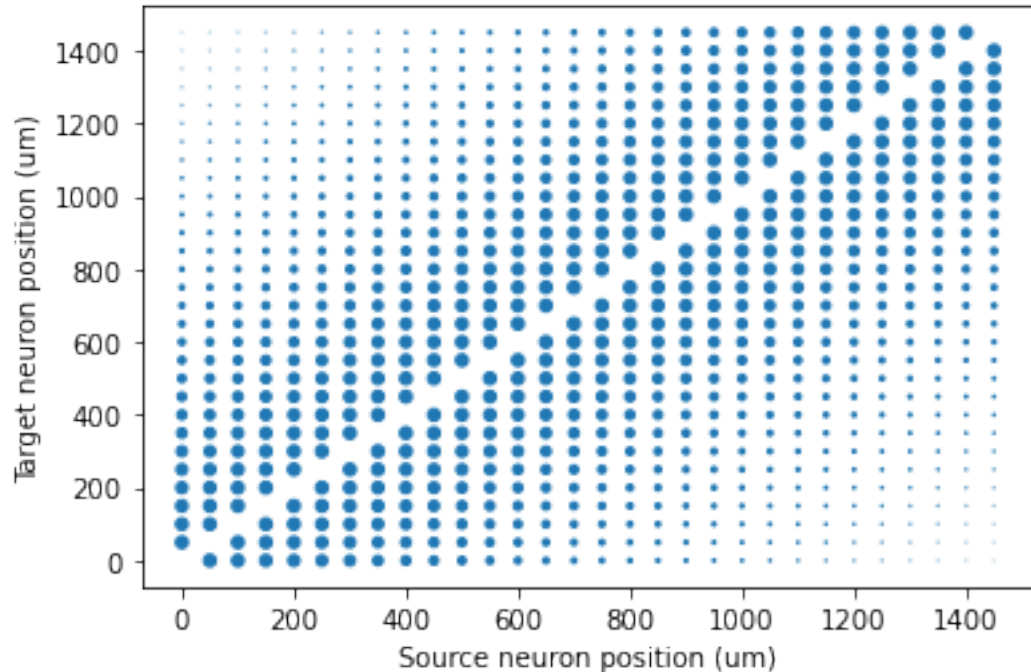
Now try changing that function and seeing how the plot changes.

2.2.5 More complex synapse models: STDP

Brian's synapse framework is very general and can do things like short-term plasticity (STP) or spike-timing dependent plasticity (STDP). Let's see how that works for STDP.

STDP is normally defined by an equation something like this:

$$\Delta w = \sum_{t_{pre}} \sum_{t_{post}} W(t_{post} - t_{pre})$$

That is, the change in synaptic weight w is the sum over all presynaptic spike times t_{pre} and postsynaptic spike times t_{post} of some function W of the difference in these spike times. A commonly used function W is:

$$W(\Delta t) = \begin{cases} A_{pre}e^{-\Delta t/\tau_{pre}} & \Delta t > 0 \\ A_{post}e^{\Delta t/\tau_{post}} & \Delta t < 0 \end{cases}$$

This function looks like this:

```
tau_pre = tau_post = 20*ms
A_pre = 0.01
A_post = -A_pre*1.05
delta_t = linspace(-50, 50, 100)*ms
W = where(delta_t>0, A_pre*exp(-delta_t/tau_pre), A_post*exp(delta_t/tau_post))
plot(delta_t/ms, W)
xlabel(r'$\Delta t$ (ms)')
ylabel('W')
axhline(0, ls='-', c='k');
```

```
<matplotlib.lines.Line2D at 0x7fdccb5acdd0>
```
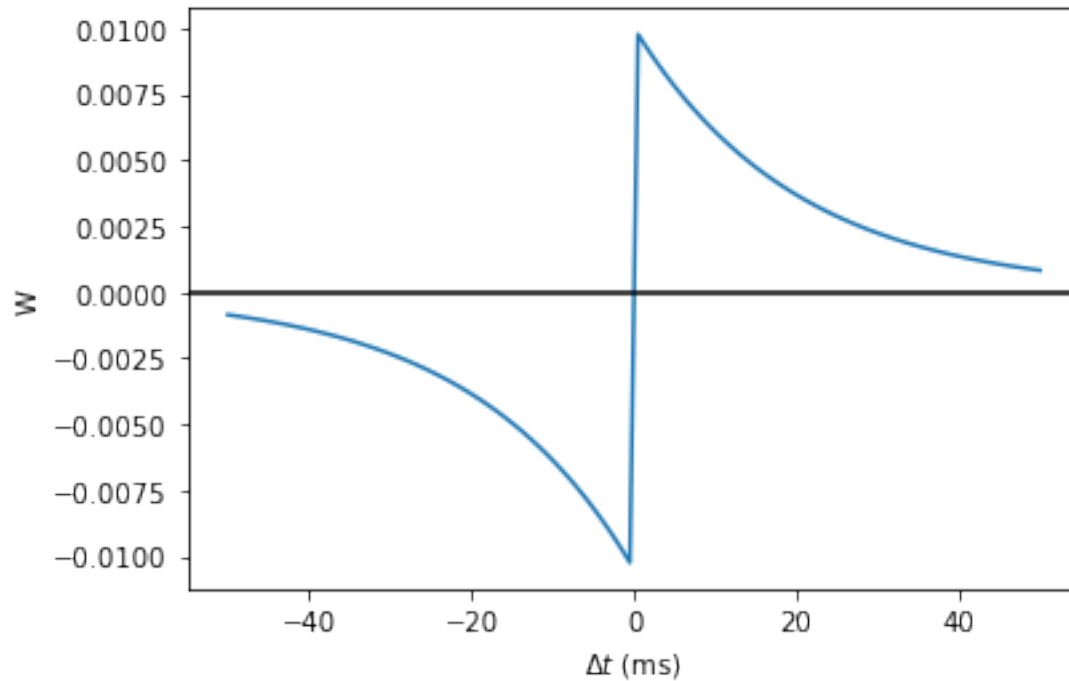
Simulating it directly using this equation though would be very inefficient, because we would have to sum over all pairs of spikes. That would also be physiologically unrealistic because the neuron cannot remember all its previous spike times. It turns out there is a more efficient and physiologically more plausible way to get the same effect.

We define two new variables a_{pre} and a_{post} which are "traces" of pre- and post-synaptic activity, governed by the differential equations:

$$\tau_{pre} \frac{\mathrm{d}}{\mathrm{d}t} a_{pre} = -a_{pre}$$
$$\tau_{post} \frac{\mathrm{d}}{\mathrm{d}t} a_{post} = -a_{post}$$

When a presynaptic spike occurs, the presynaptic trace is updated and the weight is modified according to the rule:

$$a_{pre} \to a_{pre} + A_{pre}$$
$$w \to w + a_{post}$$

When a postsynaptic spike occurs:

$$a_{post} \to a_{post} + A_{post}$$
$$w \to w + a_{pre}$$

To see that this formulation is equivalent, you just have to check that the equations sum linearly, and consider two cases: what happens if the presynaptic spike occurs before the postsynaptic spike, and vice versa. Try drawing a picture of it.

Now that we have a formulation that relies only on differential equations and spike events, we can turn that into Brian code.

```
start_scope()

taupre = taupost = 20*ms
wmax = 0.01
Apre = 0.01
```

```
Apost = -Apre*taupre/taupost*1.05

G = NeuronGroup(1, 'v:1', threshold='v>1', reset='')

S = Synapses(G, G,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (event-driven)
             dapost/dt = -apost/taupost : 1 (event-driven)
             ''',
             on_pre='''
             v_post += w
             apre += Apre
             w = clip(w+apost, 0, wmax)
             ''',
             on_post='''
             apost += Apost
             w = clip(w+apre, 0, wmax)
             ''')
```

There are a few things to see there. Firstly, when defining the synapses we've given a more complicated multi-line string
defining three synaptic variables (`w`, `apre` and `apost`). We've also got a new bit of syntax there, `(event-driven)`
after the definitions of `apre` and `apost`. What this means is that although these two variables evolve continuously over
time, Brian should only update them at the time of an event (a spike). This is because we don't need the values of `apre`
and `apost` except at spike times, and it is more efficient to only update them when needed.

Next we have a `on_pre=...` argument. The first line is `v_post += w`: this is the line that actually applies the
synaptic weight to the target neuron. The second line is `apre += Apre` which encodes the rule above. In the third
line, we're also encoding the rule above but we've added one extra feature: we've clamped the synaptic weights between
a minimum of 0 and a maximum of `wmax` so that the weights can't get too large or negative. The function `clip(x,
low, high)` does this.

Finally, we have a `on_post=...` argument. This gives the statements to calculate when a post-synaptic neuron fires.
Note that we do not modify `v` in this case, only the synaptic variables.

Now let's see how all the variables behave when a presynaptic spike arrives some time before a postsynaptic spike.

```
start_scope()

taupre = taupost = 20*ms
wmax = 0.01
Apre = 0.01
Apost = -Apre*taupre/taupost*1.05

G = NeuronGroup(2, 'v:1', threshold='t>(1+i)*10*ms', refractory=100*ms)

S = Synapses(G, G,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (clock-driven)
             dapost/dt = -apost/taupost : 1 (clock-driven)
             ''',
             on_pre='''
             v_post += w
             apre += Apre
             w = clip(w+apost, 0, wmax)
             ''',
```

```
            on_post='''
            apost += Apost
            w = clip(w+apre, 0, wmax)
            ''', method='linear')
S.connect(i=0, j=1)
M = StateMonitor(S, ['w', 'apre', 'apost'], record=True)

run(30*ms)

figure(figsize=(4, 8))
subplot(211)
plot(M.t/ms, M.apre[0], label='apre')
plot(M.t/ms, M.apost[0], label='apost')
legend()
subplot(212)
plot(M.t/ms, M.w[0], label='w')
legend(loc='best')
xlabel('Time (ms)');
```

```
Text(0.5, 0, 'Time (ms)')
```
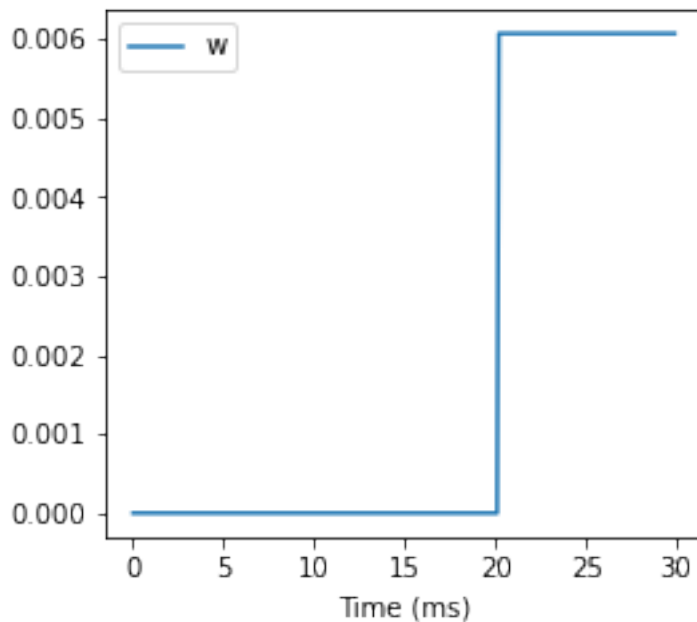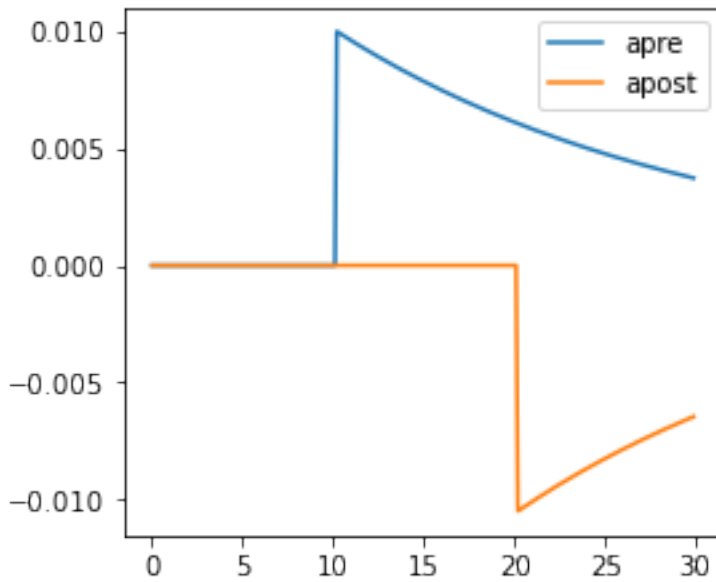
A couple of things to note here. First of all, we've used a trick to make neuron 0 fire a spike at time 10 ms, and neuron 1 at time 20 ms. Can you see how that works?

Secondly, we've replaced the `(event-driven)` by `(clock-driven)` so you can see how `apre` and `apost` evolve over time. Try reverting this change and see what happens.

Try changing the times of the spikes to see what happens.

Finally, let's verify that this formulation is equivalent to the original one.

```
start_scope()

taupre = taupost = 20*ms
Apre = 0.01
```

(continues on next page)

```python
Apost = -Apre*taupre/taupost*1.05
tmax = 50*ms
N = 100

# Presynaptic neurons G spike at times from 0 to tmax
# Postsynaptic neurons G spike at times from tmax to 0
# So difference in spike times will vary from -tmax to +tmax
G = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
H = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
G.tspike = 'i*tmax/(N-1)'
H.tspike = '(N-1-i)*tmax/(N-1)'

S = Synapses(G, H,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (event-driven)
             dapost/dt = -apost/taupost : 1 (event-driven)
             ''',
             on_pre='''
             apre += Apre
             w = w+apost
             ''',
             on_post='''
             apost += Apost
             w = w+apre
             ''')
S.connect(j='i')

run(tmax+1*ms)

plot((H.tspike-G.tspike)/ms, S.w)
xlabel(r'$\Delta t$ (ms)')
ylabel(r'$\Delta w$')
axhline(0, ls='-', c='k');
```

```
<matplotlib.lines.Line2D at 0x7fdcc8ae8890>
```
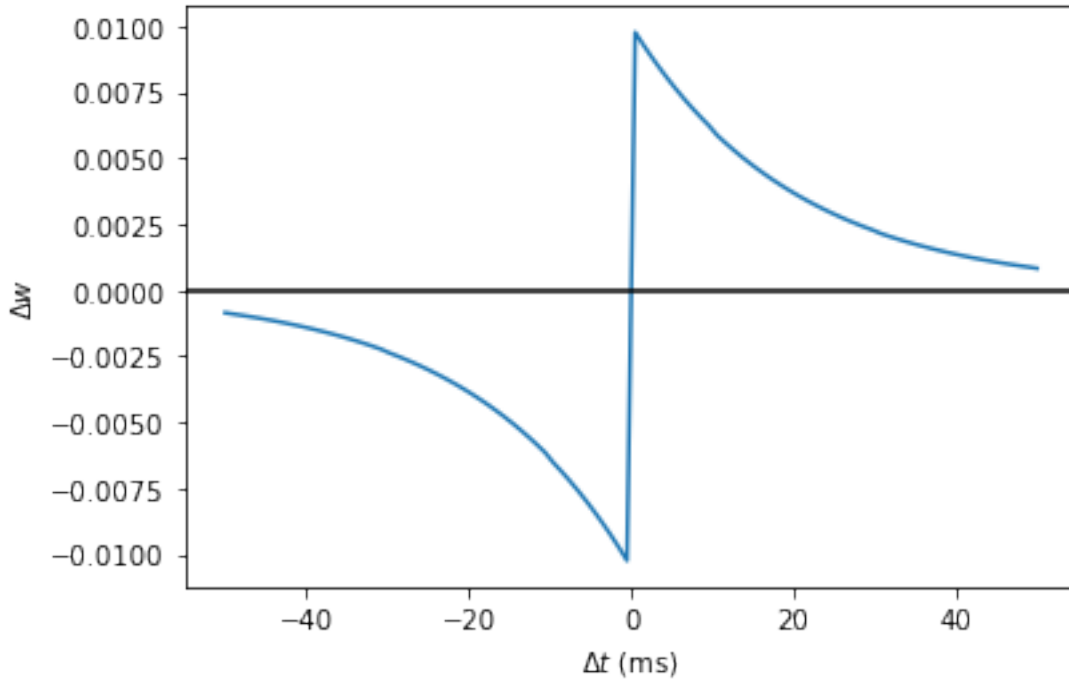
Can you see how this works?

2.2.6 End of tutorial

2.3 Introduction to Brian part 3: Simulations

If you haven't yet read parts 1 and 2 on Neurons and Synapses, go read them first.

This tutorial is about managing the slightly more complicated tasks that crop up in research problems, rather than the toy examples we've been looking at so far. So we cover things like inputting sensory data, modelling experimental conditions, etc.

As before we start by importing the Brian package and setting up matplotlib for IPython:

```
from brian2 import *
%matplotlib inline
```

2.3.1 Multiple runs

Let's start by looking at a very common task: doing multiple runs of a simulation with some parameter that changes. Let's start off with something very simple, how does the firing rate of a leaky integrate-and-fire neuron driven by Poisson spiking neurons change depending on its membrane time constant? Let's set that up.

```
# remember, this is here for running separate simulations in the same notebook
start_scope()
# Parameters
num_inputs = 100
input_rate = 10*Hz
weight = 0.1
```

```python
# Range of time constants
tau_range = linspace(1, 10, 30)*ms
# Use this list to store output rates
output_rates = []
# Iterate over range of time constants
for tau in tau_range:
    # Construct the network each time
    P = PoissonGroup(num_inputs, rates=input_rate)
    eqs = '''
    dv/dt = -v/tau : 1
    '''
    G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='exact')
    S = Synapses(P, G, on_pre='v += weight')
    S.connect()
    M = SpikeMonitor(G)
    # Run it and store the output firing rate in the list
    run(1*second)
    output_rates.append(M.num_spikes/second)
# And plot it
plot(tau_range/ms, output_rates)
xlabel(r'$\tau$ (ms)')
ylabel('Firing rate (sp/s)');
```
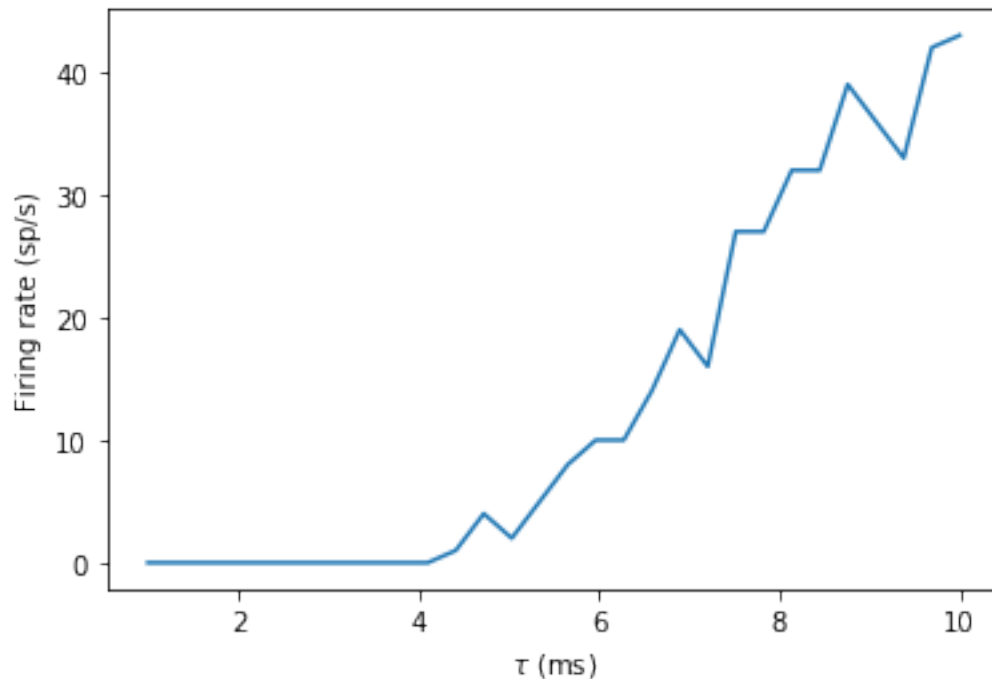

Now if you're running the notebook, you'll see that this was a little slow to run. The reason is that for each loop, you're recreating the objects from scratch. We can improve that by setting up the network just once. We store a copy of the state of the network before the loop, and restore it at the beginning of each iteration.

```python
start_scope()
num_inputs = 100
input_rate = 10*Hz
weight = 0.1
tau_range = linspace(1, 10, 30)*ms
```

```python
output_rates = []
# Construct the network just once
P = PoissonGroup(num_inputs, rates=input_rate)
eqs = '''
dv/dt = -v/tau : 1
'''
G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='exact')
S = Synapses(P, G, on_pre='v += weight')
S.connect()
M = SpikeMonitor(G)
# Store the current state of the network
store()
for tau in tau_range:
    # Restore the original state of the network
    restore()
    # Run it with the new value of tau
    run(1*second)
    output_rates.append(M.num_spikes/second)
plot(tau_range/ms, output_rates)
xlabel(r'$\tau$ (ms)')
ylabel('Firing rate (sp/s)');
```
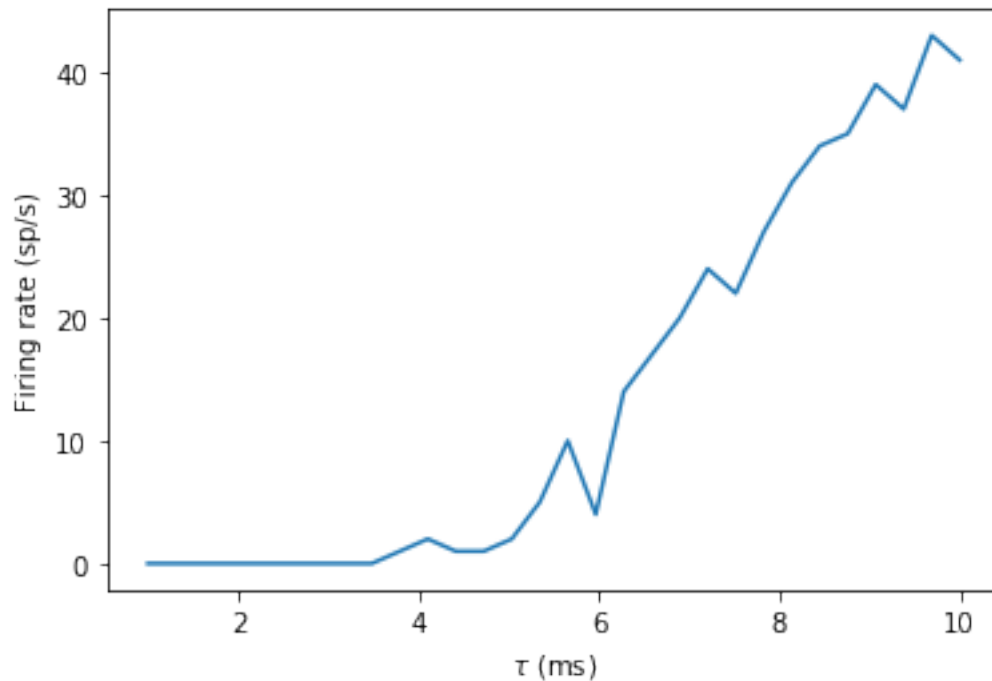

That's a very simple example of using store and restore, but you can use it in much more complicated situations. For example, you might want to run a long training run, and then run multiple test runs afterwards. Simply put a store after the long training run, and a restore before each testing run.

You can also see that the output curve is very noisy and doesn't increase monotonically like we'd expect. The noise is coming from the fact that we run the Poisson group afresh each time. If we only wanted to see the effect of the time constant, we could make sure that the spikes were the same each time (although note that really, you ought to do multiple runs and take an average). We do this by running just the Poisson group once, recording its spikes, and then creating a new `SpikeGeneratorGroup` that will output those recorded spikes each time.

```
start_scope()
num_inputs = 100
input_rate = 10*Hz
weight = 0.1
tau_range = linspace(1, 10, 30)*ms
output_rates = []
# Construct the Poisson spikes just once
P = PoissonGroup(num_inputs, rates=input_rate)
MP = SpikeMonitor(P)
# We use a Network object because later on we don't
# want to include these objects
net = Network(P, MP)
net.run(1*second)
# And keep a copy of those spikes
spikes_i = MP.i
spikes_t = MP.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG = SpikeGeneratorGroup(num_inputs, spikes_i, spikes_t)
eqs = '''
dv/dt = -v/tau : 1
'''
G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='exact')
S = Synapses(SGG, G, on_pre='v += weight')
S.connect()
M = SpikeMonitor(G)
# Store the current state of the network
net = Network(SGG, G, S, M)
net.store()
for tau in tau_range:
    # Restore the original state of the network
    net.restore()
    # Run it with the new value of tau
    net.run(1*second)
    output_rates.append(M.num_spikes/second)
plot(tau_range/ms, output_rates)
xlabel(r'$\tau$ (ms)')
ylabel('Firing rate (sp/s)');
```
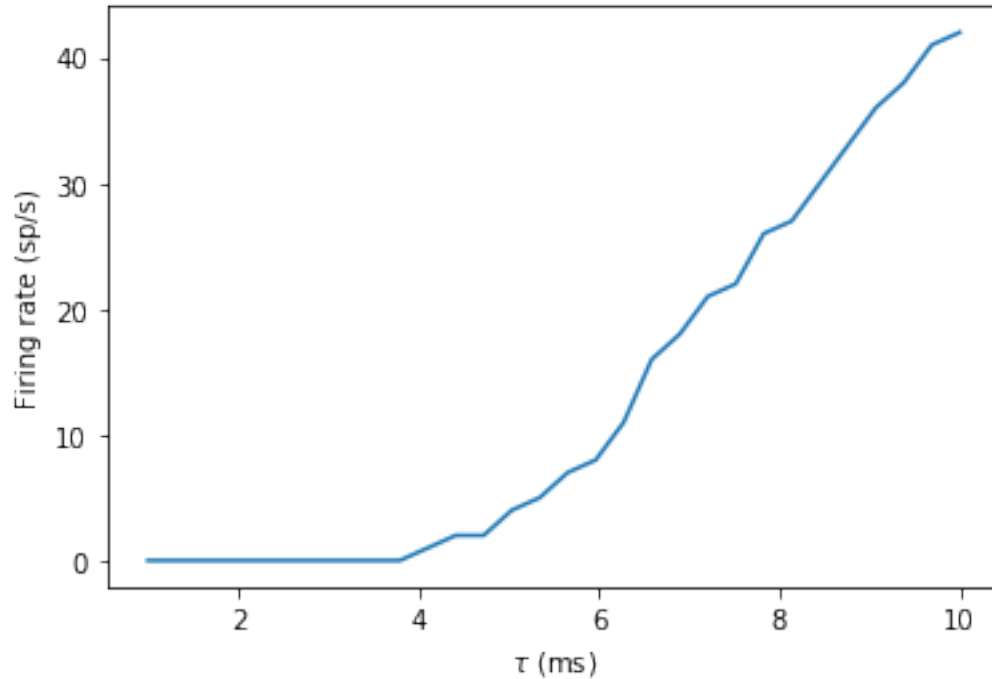
You can see that now there is much less noise and it increases monotonically because the input spikes are the same each time, meaning we're seeing the effect of the time constant, not the random spikes.

Note that in the code above, we created `Network` objects. The reason is that in the loop, if we just called `run` it would try to simulate all the objects, including the Poisson neurons `P`, and we only want to run that once. We use `Network` to specify explicitly which objects we want to include.

The techniques we've looked at so far are the conceptually most simple way to do multiple runs, but not always the most efficient. Since there's only a single output neuron in the model above, we can simply duplicate that output neuron and make the time constant a parameter of the group.

```
start_scope()
num_inputs = 100
input_rate = 10*Hz
weight = 0.1
tau_range = linspace(1, 10, 30)*ms
num_tau = len(tau_range)
P = PoissonGroup(num_inputs, rates=input_rate)
# We make tau a parameter of the group
eqs = '''
dv/dt = -v/tau : 1
tau : second
'''
# And we have num_tau output neurons, each with a different tau
G = NeuronGroup(num_tau, eqs, threshold='v>1', reset='v=0', method='exact')
G.tau = tau_range
S = Synapses(P, G, on_pre='v += weight')
S.connect()
M = SpikeMonitor(G)
# Now we can just run once with no loop
run(1*second)
output_rates = M.count/second # firing rate is count/duration
plot(tau_range/ms, output_rates)
```

```
xlabel(r'$\tau$ (ms)')
ylabel('Firing rate (sp/s)');
```

```
WARNING    "tau" is an internal variable of group "neurongroup", but also exists in␣
→the run namespace with the value 10. * msecond. The internal variable will be used.␣
→[brian2.groups.group.Group.resolve.resolution_conflict]
```
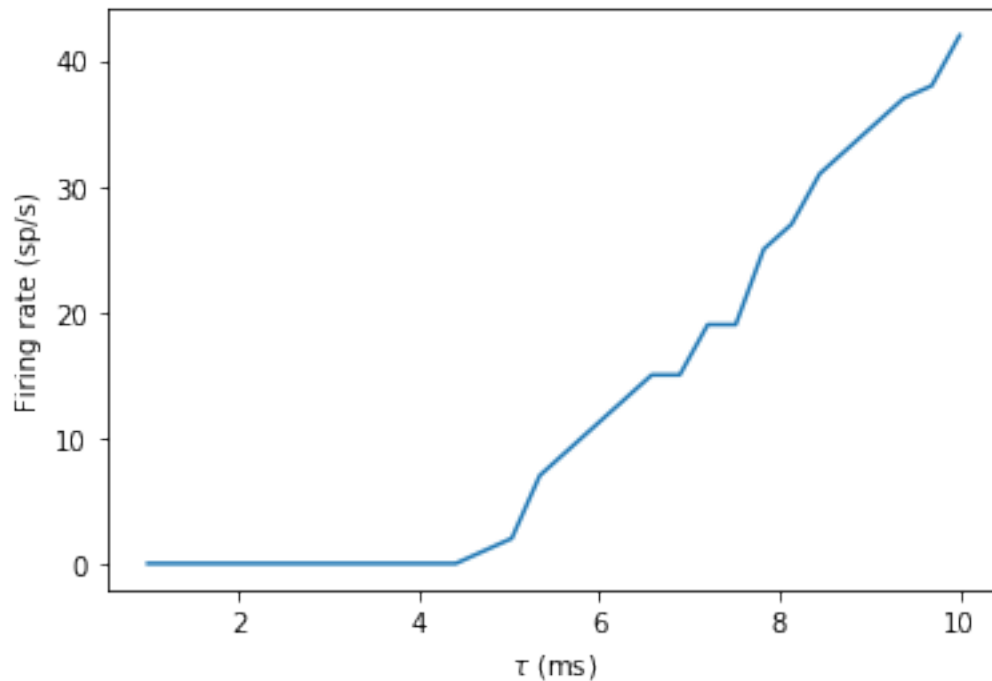

You can see that this is much faster again! It's a little bit more complicated conceptually, and it's not always possible to do this trick, but it can be much more efficient if it's possible.

Let's finish with this example by having a quick look at how the mean and standard deviation of the interspike intervals depends on the time constant.

```
trains = M.spike_trains()
isi_mu = full(num_tau, nan)*second
isi_std = full(num_tau, nan)*second
for idx in range(num_tau):
    train = diff(trains[idx])
    if len(train)>1:
        isi_mu[idx] = mean(train)
        isi_std[idx] = std(train)
errorbar(tau_range/ms, isi_mu/ms, yerr=isi_std/ms)
xlabel(r'$\tau$ (ms)')
ylabel('Interspike interval (ms)');
```
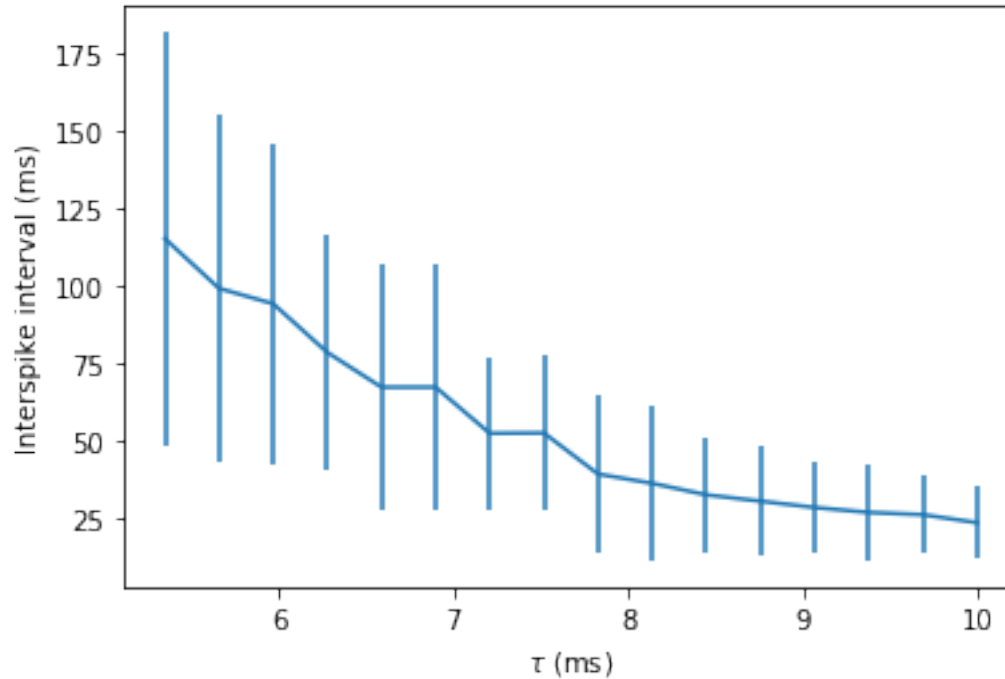
Notice that we used the `spike_trains()` method of `SpikeMonitor`. This is a dictionary with keys being the indices of the neurons and values being the array of spike times for that neuron.

2.3.2 Changing things during a run

Imagine an experiment where you inject current into a neuron, and change the amplitude randomly every 10 ms. Let's see if we can model that using a Hodgkin-Huxley type neuron.

```
start_scope()
# Parameters
area = 20000*umetre**2
Cm = 1*ufarad*cm**-2 * area
gl = 5e-5*siemens*cm**-2 * area
El = -65*mV
EK = -90*mV
ENa = 50*mV
g_na = 100*msiemens*cm**-2 * area
g_kd = 30*msiemens*cm**-2 * area
VT = -63*mV
# The model
eqs_HH = '''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/Cm : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
    (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
    (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
    (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
→ms*h : 1
I : amp
'''
group = NeuronGroup(1, eqs_HH,
```

```
                        threshold='v > -40*mV',
                        refractory='v > -40*mV',
                        method='exponential_euler')
group.v = El
statemon = StateMonitor(group, 'v', record=True)
spikemon = SpikeMonitor(group, variables='v')
figure(figsize=(9, 4))
for l in range(5):
    group.I = rand()*50*nA
    run(10*ms)
    axvline(l*10, ls='--', c='k')
axhline(El/mV, ls='-', c='lightgray', lw=3)
plot(statemon.t/ms, statemon.v[0]/mV, '-b')
plot(spikemon.t/ms, spikemon.v/mV, 'ob')
xlabel('Time (ms)')
ylabel('v (mV)');
```
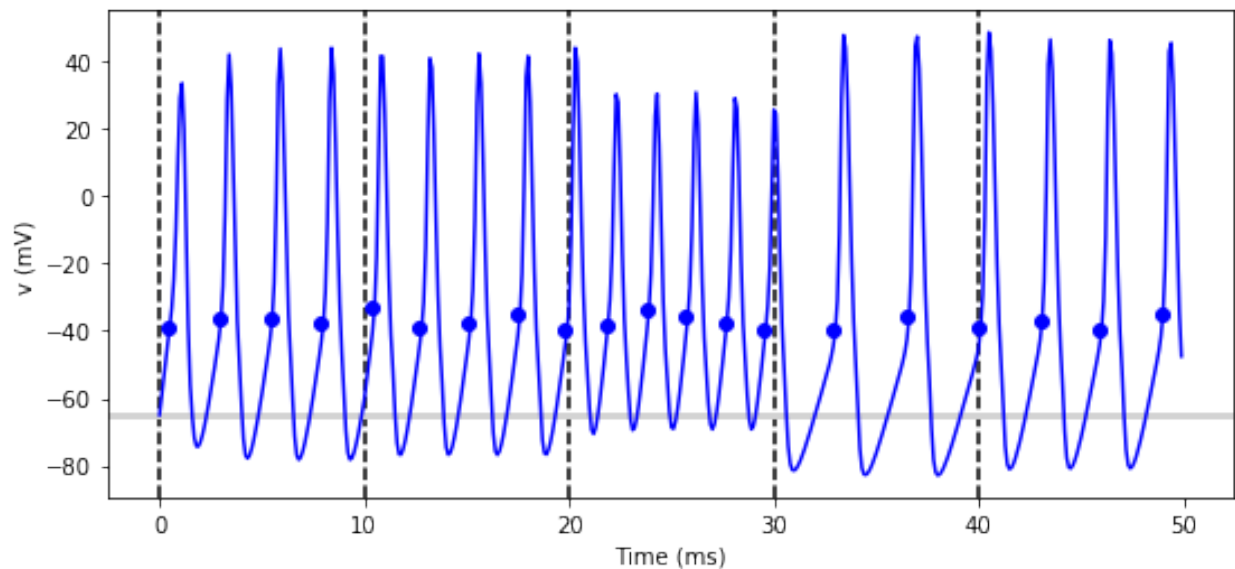

In the code above, we used a loop over multiple runs to achieve this. That's fine, but it's not the most efficient way to do it because each time we call `run` we have to do a lot of initialisation work that slows everything down. It also won't work as well with the more efficient standalone mode of Brian. Here's another way.

```
start_scope()
group = NeuronGroup(1, eqs_HH,
                        threshold='v > -40*mV',
                        refractory='v > -40*mV',
                        method='exponential_euler')
group.v = El
statemon = StateMonitor(group, 'v', record=True)
spikemon = SpikeMonitor(group, variables='v')
# we replace the loop with a run_regularly
group.run_regularly('I = rand()*50*nA', dt=10*ms)
run(50*ms)
figure(figsize=(9, 4))
# we keep the loop just to draw the vertical lines
for l in range(5):
    axvline(l*10, ls='--', c='k')
```

```
axhline(El/mV, ls='-', c='lightgray', lw=3)
plot(statemon.t/ms, statemon.v[0]/mV, '-b')
plot(spikemon.t/ms, spikemon.v/mV, 'ob')
xlabel('Time (ms)')
ylabel('v (mV)');
```
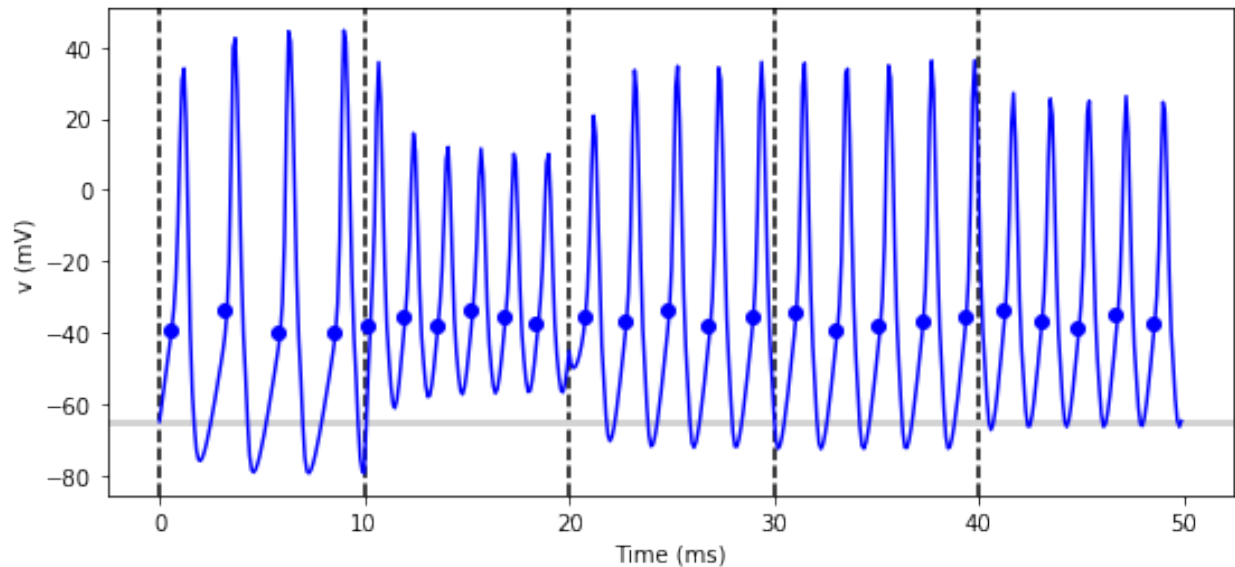

We've replaced the loop that had multiple `run` calls with a `run_regularly`. This makes the specified block of code run every `dt=10*ms`. The `run_regularly` lets you run code specific to a single `NeuronGroup`, but sometimes you might need more flexibility. For this, you can use `network_operation` which lets you run arbitrary Python code (but won't work with the standalone mode).

```
start_scope()
group = NeuronGroup(1, eqs_HH,
                    threshold='v > -40*mV',
                    refractory='v > -40*mV',
                    method='exponential_euler')
group.v = El
statemon = StateMonitor(group, 'v', record=True)
spikemon = SpikeMonitor(group, variables='v')
# we replace the loop with a network_operation
@network_operation(dt=10*ms)
def change_I():
    group.I = rand()*50*nA
run(50*ms)
figure(figsize=(9, 4))
for l in range(5):
    axvline(l*10, ls='--', c='k')
axhline(El/mV, ls='-', c='lightgray', lw=3)
plot(statemon.t/ms, statemon.v[0]/mV, '-b')
plot(spikemon.t/ms, spikemon.v/mV, 'ob')
xlabel('Time (ms)')
ylabel('v (mV)');
```
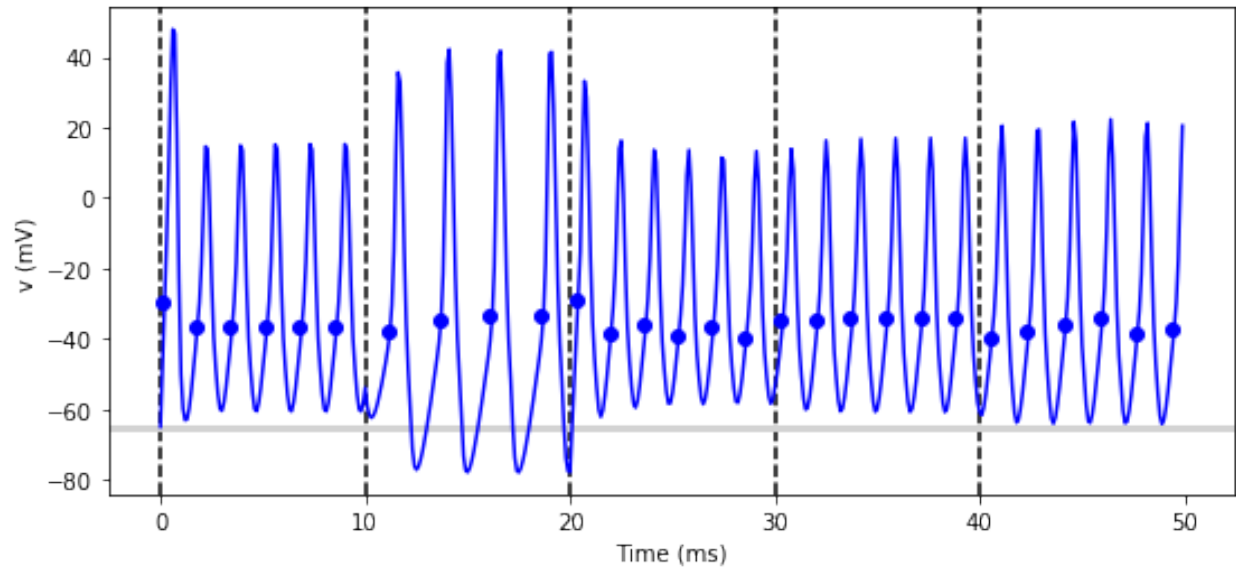
Now let's extend this example to run on multiple neurons, each with a different capacitance to see how that affects the behaviour of the cell.

```
start_scope()
N = 3
eqs_HH_2 = '''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/C : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
    (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
    (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
    (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
↪ms*h : 1
I : amp
C : farad
'''
group = NeuronGroup(N, eqs_HH_2,
                    threshold='v > -40*mV',
                    refractory='v > -40*mV',
                    method='exponential_euler')
group.v = El
# initialise with some different capacitances
group.C = array([0.8, 1, 1.2])*ufarad*cm**-2*area
statemon = StateMonitor(group, variables=True, record=True)
# we go back to run_regularly
group.run_regularly('I = rand()*50*nA', dt=10*ms)
run(50*ms)
figure(figsize=(9, 4))
for l in range(5):
    axvline(l*10, ls='--', c='k')
axhline(El/mV, ls='-', c='lightgray', lw=3)
plot(statemon.t/ms, statemon.v.T/mV, '-')
xlabel('Time (ms)')
ylabel('v (mV)');
```
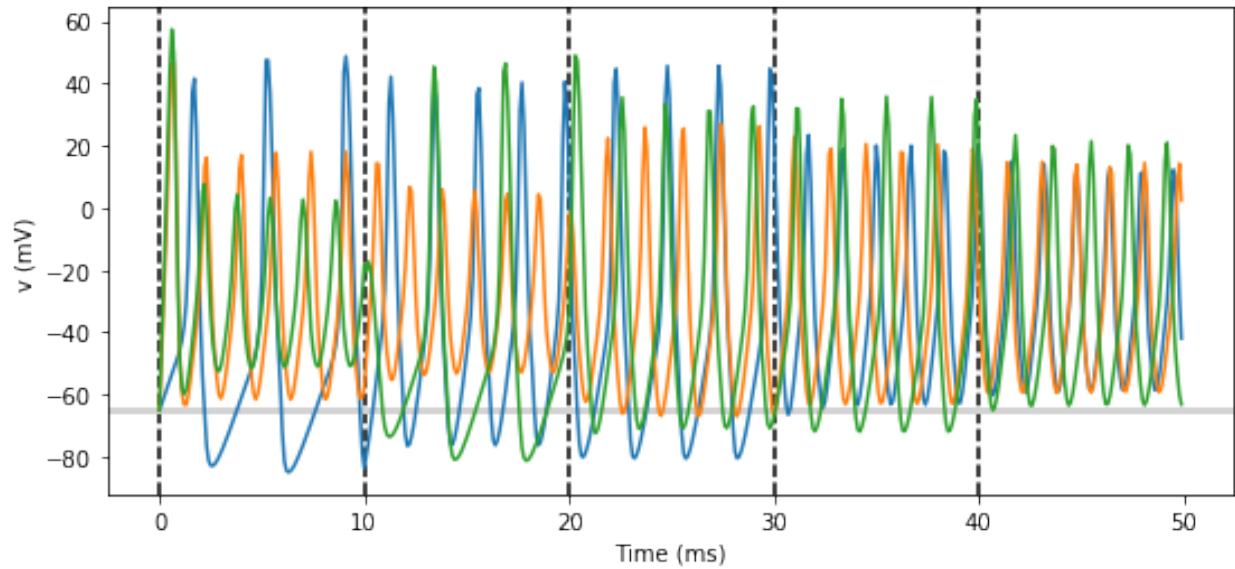
So that runs, but something looks wrong! The injected currents look like they're different for all the different neurons! Let's check:

```
plot(statemon.t/ms, statemon.I.T/nA, '-')
xlabel('Time (ms)')
ylabel('I (nA)');
```
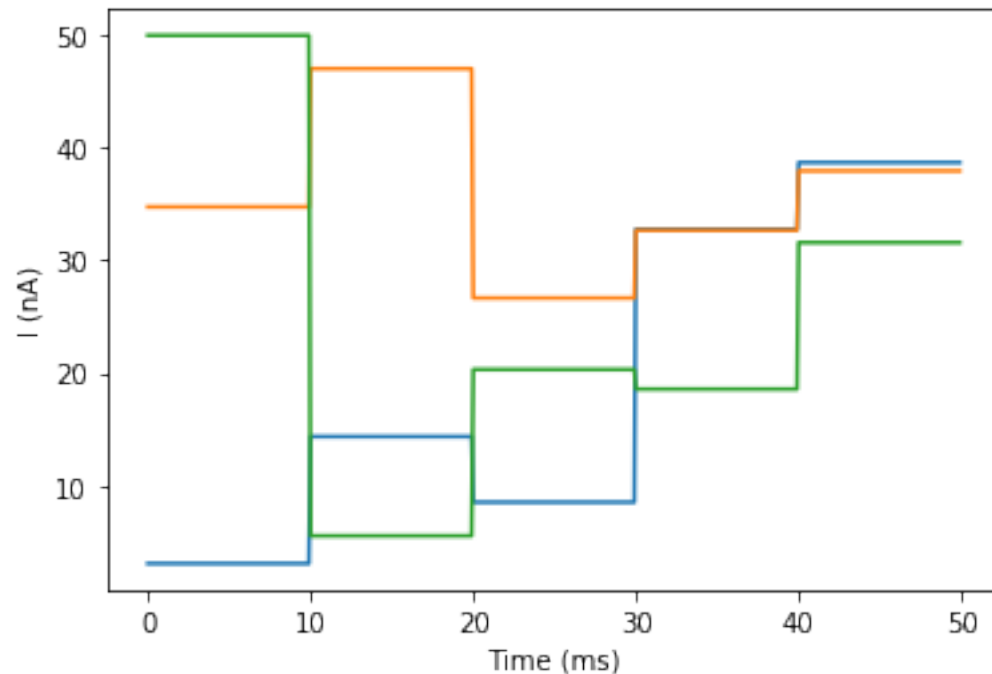

Sure enough, it's different each time. But why? We wrote `group.run_regularly('I = rand()*50*nA', dt=10*ms)` which seems like it should give the same value of I for each neuron. But, like threshold and reset statements, `run_regularly` code is interpreted as being run separately for each neuron, and because I is a parameter, it can be different for each neuron. We can fix this by making I into a *shared* variable, meaning it has the same value for each neuron.

```
start_scope()
N = 3
eqs_HH_3 = '''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/C : volt
dm/dt = 0.32*(mV**-1)*(13.*mV-v+VT)/
    (exp((13.*mV-v+VT)/(4.*mV))-1.)/ms*(1-m)-0.28*(mV**-1)*(v-VT-40.*mV)/
    (exp((v-VT-40.*mV)/(5.*mV))-1.)/ms*m : 1
dn/dt = 0.032*(mV**-1)*(15.*mV-v+VT)/
    (exp((15.*mV-v+VT)/(5.*mV))-1.)/ms*(1.-n)-.5*exp((10.*mV-v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
→ms*h : 1
I : amp (shared) # everything is the same except we've added this shared
C : farad
'''
group = NeuronGroup(N, eqs_HH_3,
                    threshold='v > -40*mV',
                    refractory='v > -40*mV',
                    method='exponential_euler')
group.v = El
group.C = array([0.8, 1, 1.2])*ufarad*cm**-2*area
statemon = StateMonitor(group, 'v', record=True)
group.run_regularly('I = rand()*50*nA', dt=10*ms)
run(50*ms)
figure(figsize=(9, 4))
for l in range(5):
    axvline(l*10, ls='--', c='k')
axhline(El/mV, ls='-', c='lightgray', lw=3)
plot(statemon.t/ms, statemon.v.T/mV, '-')
xlabel('Time (ms)')
ylabel('v (mV)');
```
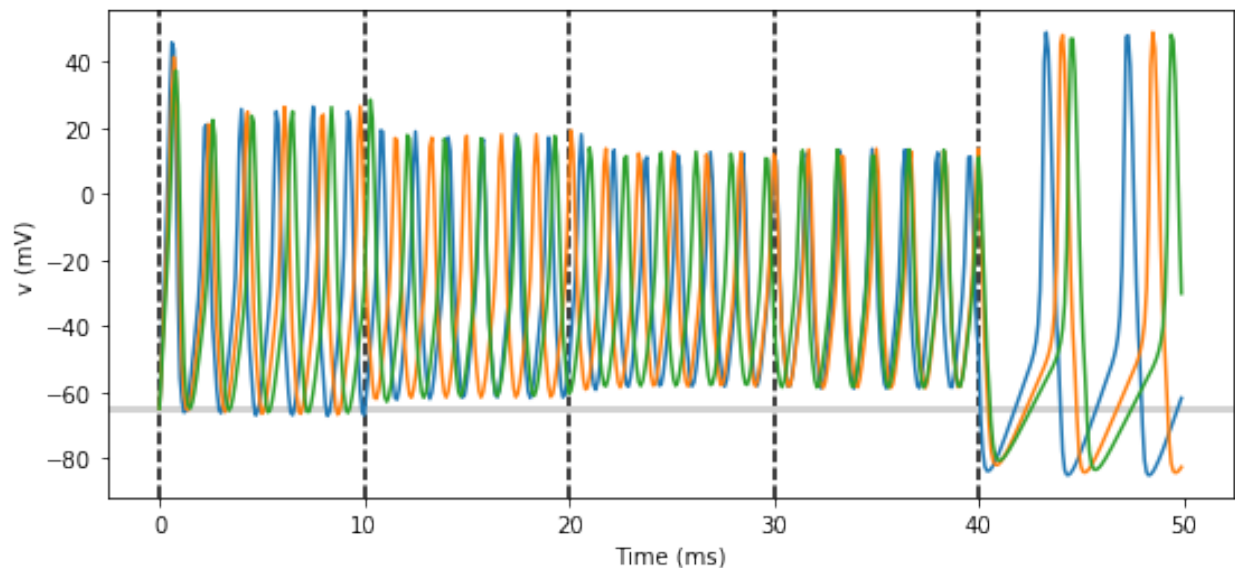

Ahh, that's more like it!

2.3.3 Adding input

Now let's think about a neuron being driven by a sinusoidal input. Let's go back to a leaky integrate-and-fire to simplify the equations a bit.

```
start_scope()
A = 2.5
f = 10*Hz
tau = 5*ms
eqs = '''
dv/dt = (I-v)/tau : 1
I = A*sin(2*pi*f*t) : 1
'''
G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='euler')
M = StateMonitor(G, variables=True, record=True)
run(200*ms)
plot(M.t/ms, M.v[0], label='v')
plot(M.t/ms, M.I[0], label='I')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best');
```
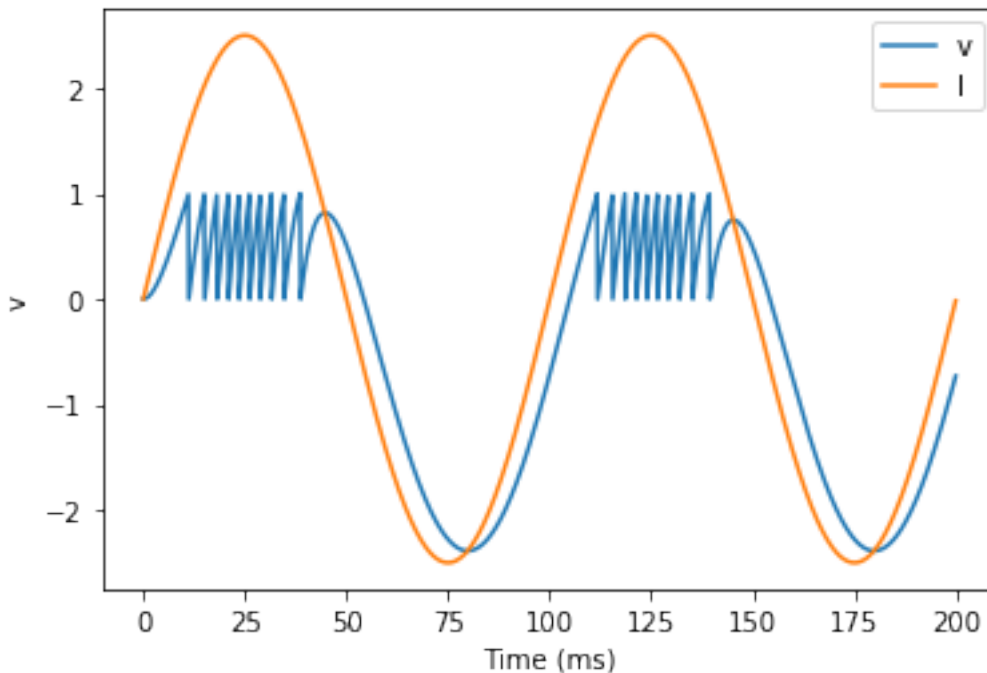

So far, so good and the sort of thing we saw in the first tutorial. Now, what if that input current were something we had recorded and saved in a file? In that case, we can use `TimedArray`. Let's start by reproducing the picture above but using `TimedArray`.

```
start_scope()
A = 2.5
f = 10*Hz
tau = 5*ms
# Create a TimedArray and set the equations to use it
t_recorded = arange(int(200*ms/defaultclock.dt))*defaultclock.dt
I_recorded = TimedArray(A*sin(2*pi*f*t_recorded), dt=defaultclock.dt)
```

```
eqs = '''
dv/dt = (I-v)/tau : 1
I = I_recorded(t) : 1
'''
G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='exact')
M = StateMonitor(G, variables=True, record=True)
run(200*ms)
plot(M.t/ms, M.v[0], label='v')
plot(M.t/ms, M.I[0], label='I')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best');
```
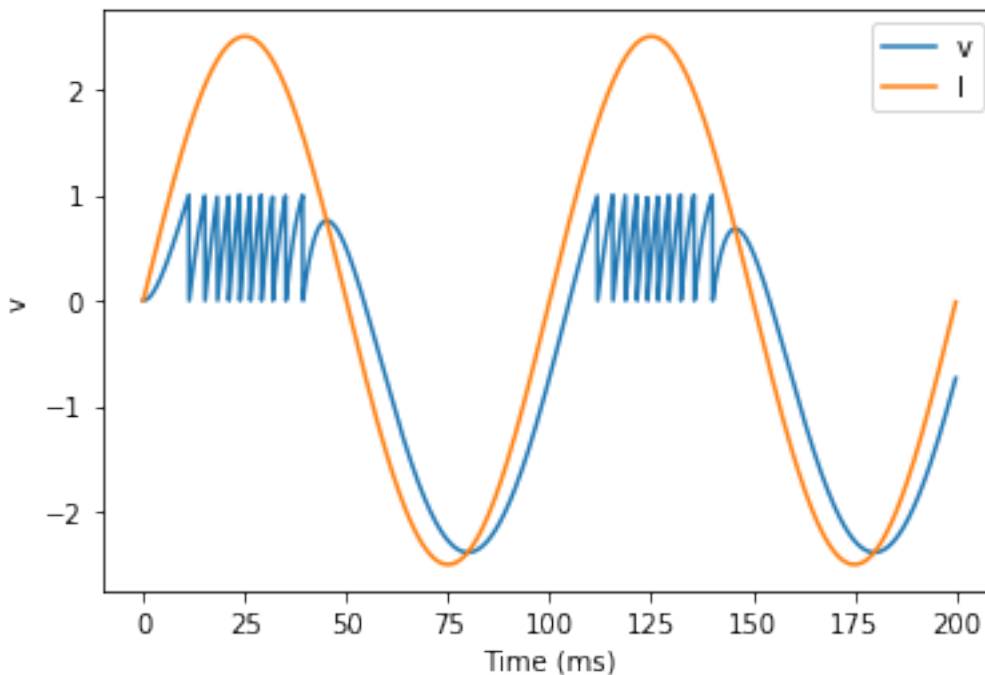

Note that for the example where we put the `sin` function directly in the equations, we had to use the `method='euler'` argument because the exact integrator wouldn't work here (try it!). However, `TimedArray` is considered to be constant over its time step and so the linear integrator can be used. This means you won't get the same behaviour from these two methods for two reasons. Firstly, the numerical integration methods `exact` and `euler` give slightly different results. Secondly, `sin` is not constant over a timestep whereas `TimedArray` is.

Now just to show that `TimedArray` works for arbitrary currents, let's make a weird "recorded" current and run it on that.

```
start_scope()
A = 2.5
f = 10*Hz
tau = 5*ms
# Let's create an array that couldn't be
# reproduced with a formula
num_samples = int(200*ms/defaultclock.dt)
I_arr = zeros(num_samples)
for _ in range(100):
    a = randint(num_samples)
```

```
    I_arr[a:a+100] = rand()
I_recorded = TimedArray(A*I_arr, dt=defaultclock.dt)
eqs = '''
dv/dt = (I-v)/tau : 1
I = I_recorded(t) : 1
'''
G = NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='exact')
M = StateMonitor(G, variables=True, record=True)
run(200*ms)
plot(M.t/ms, M.v[0], label='v')
plot(M.t/ms, M.I[0], label='I')
xlabel('Time (ms)')
ylabel('v')
legend(loc='best');
```
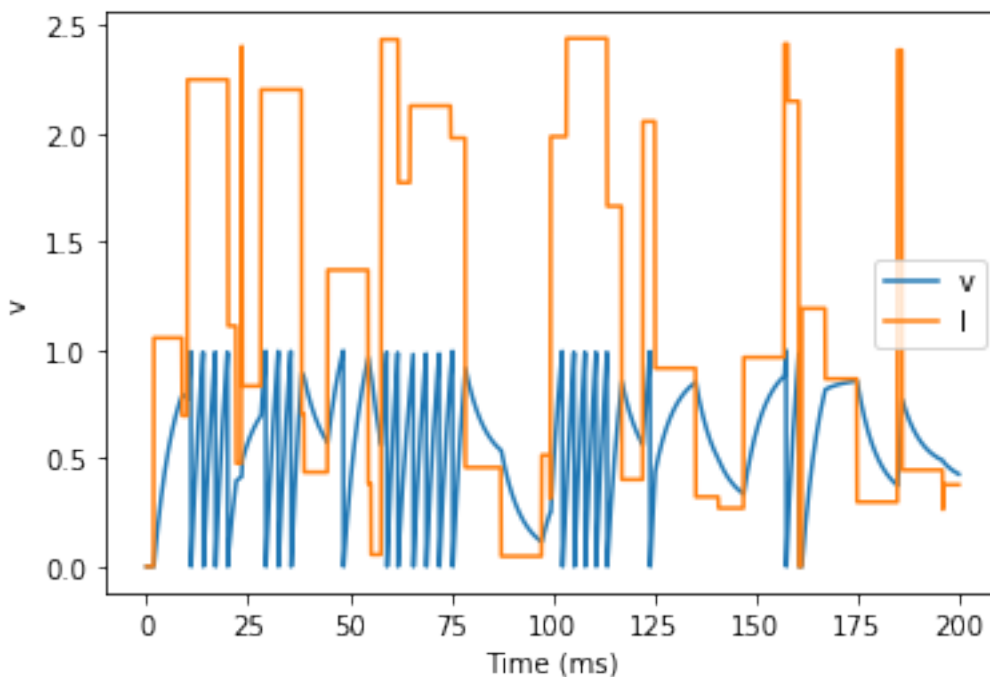

Finally, let's finish on an example that actually reads in some data from a file. See if you can work out how this example works.

```
start_scope()
from matplotlib.image import imread
img = (1-imread('brian.png'))[::-1, :, 0].T
num_samples, N = img.shape
ta = TimedArray(img, dt=1*ms) # 228
A = 1.5
tau = 2*ms
eqs = '''
dv/dt = (A*ta(t, i)-v)/tau+0.8*xi*tau**-0.5 : 1
'''
G = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', method='euler')
M = SpikeMonitor(G)
run(num_samples*ms)
plot(M.t/ms, M.i, '.k', ms=3)
```

```
xlim(0, num_samples)
ylim(0, N)
xlabel('Time (ms)')
ylabel('Neuron index');
```

USER'S GUIDE
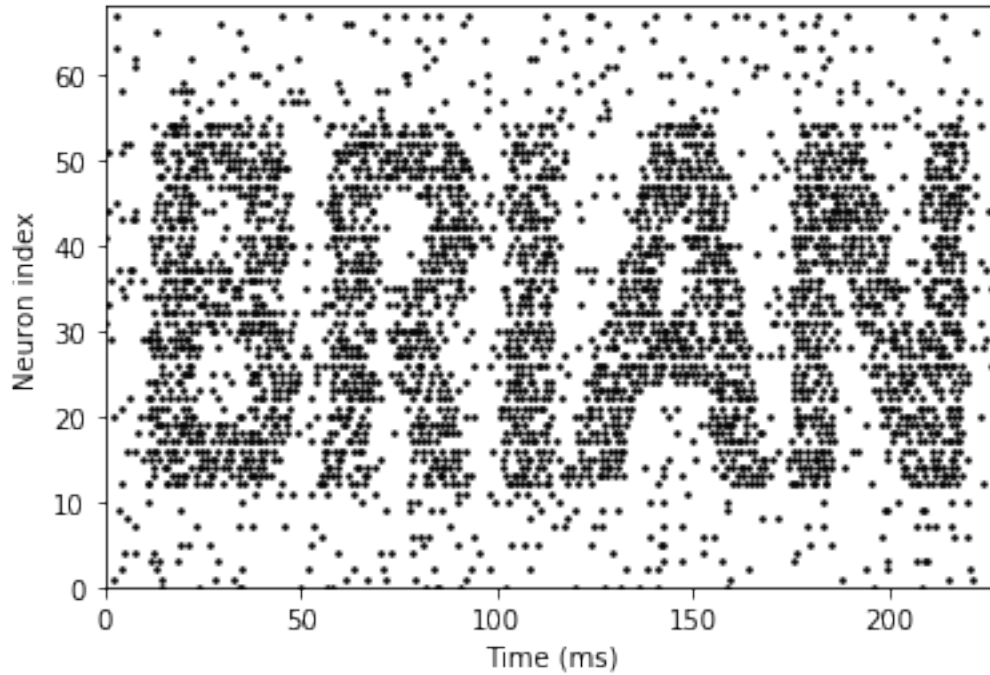
3.1 Importing Brian

After installation, Brian is available in the `brian2` package. By doing a wildcard import from this package, i.e.:

```
from brian2 import *
```

you will not only get access to the `brian2` classes and functions, but also to everything in the `pylab` package, which includes the plotting functions from matplotlib and everything included in numpy/scipy (e.g. functions such as `arange`, `linspace`, etc.). Apart from this when you use the wildcard import, the builtin `input` function is overshadowed by the `input` module in the `brian2` package. If you wish to use the builtin `input` function in your program after importing the brian2 package then you can explicitly import the `input` function again as shown below:

```
from brian2 import *
from builtins import input
```

The following topics are not essential for beginners.

3.1.1 Precise control over importing

If you want to use a wildcard import from Brian, but don't want to import all the additional symbols provided by `pylab` or don't want to overshadow the builtin `input` function, you can use:

```
from brian2.only import *
```

Note that whenever you use something different from the most general `from brian2 import *` statement, you should be aware that Brian overwrites some numpy functions with their unit-aware equivalents (see *Units*). If you combine multiple wildcard imports, the Brian import should therefore be the last import. Similarly, you should not import and call overwritten numpy functions directly, e.g. by using `import numpy as np` followed by `np.sin` since this will not use the unit-aware versions. To make this easier, Brian provides a `brian2.numpy_` package that provides access to everything in numpy but overwrites certain functions. If you prefer to use prefixed names, the recommended way of doing the imports is therefore:

```
import brian2.numpy_ as np
import brian2.only as br2
```

Note that it is safe to use e.g. `np.sin` and `numpy.sin` after a `from brian2 import *`.

3.1.2 Dependency checks

Brian will check the dependency versions during import and raise an error for an outdated dependency. An outdated dependency does not necessarily mean that Brian cannot be run with it, it only means that Brian is untested on that version. If you want to force Brian to run despite the outdated dependency, set the *core.outdated_dependency_error* preference to `False`. Note that this cannot be done in a script, since you do not have access to the preferences before importing `brian2`. See *Preferences* for instructions how to set preferences in a file.

3.2 Physical units

- *Using units*
- *Removing units*
- *Temperatures*
- *Constants*
- *Importing units*
- *In-place operations on quantities*

Brian includes a system for physical units. The base units are defined by their standard SI unit names: `amp`/`ampere`, `kilogram`/`kilogramme`, `second`, `metre`/`meter`, `mole`/`mol`, `kelvin`, and `candela`. In addition to these base units, Brian defines a set of derived units: `coulomb`, `farad`, `gram`/`gramme`, `hertz`, `joule`, `liter`/`litre`, `molar`, `pascal`, `ohm`, `siemens`, `volt`, `watt`, together with prefixed versions (e.g. `msiemens = 0.001*siemens`) using the prefixes `p`, `n`, `u`, `m`, `k`, `M`, `G`, `T` (two exceptions to this rule: `kilogram` is not defined with any additional prefixes, and `metre` and `meter` are additionally defined with the "centi" prefix, i.e. `cmetre`/`cmeter`). For convenience, a couple of additional useful standard abbreviations such as `cm` (instead of `cmetre`/`cmeter`), `nS` (instead of `nsiemens`), `ms` (instead of `msecond`), `Hz` (instead of `hertz`), `mM` (instead of `mmolar`) are included. To avoid clashes with common variable names, no one-letter abbreviations are provided (e.g. you can use `mV` or `nS`, but *not* `V` or `S`).

3.2.1 Using units

You can generate a physical quantity by multiplying a scalar or vector value with its physical unit:

```
>>> tau = 20*ms
>>> print(tau)
20. ms
>>> rates = [10, 20, 30]*Hz
>>> print(rates)
[ 10.  20.  30.] Hz
```

Brian will check the consistency of operations on units and raise an error for dimensionality mismatches:

```
>>> tau += 1  # ms? second?
Traceback (most recent call last):
...
DimensionMismatchError: Cannot calculate ... += 1, units do not match (units are
→second and 1).
```

```
>>> 3*kgram + 3*amp
Traceback (most recent call last):
...
DimensionMismatchError: Cannot calculate 3. kg + 3. A, units do not match (units are
→kilogram and amp).
```

Most Brian functions will also complain about non-specified or incorrect units:

```
>>> G = NeuronGroup(10, 'dv/dt = -v/tau: volt', dt=0.5)
Traceback (most recent call last):
...
DimensionMismatchError: Function "__init__" expected a quantity with unit second
→for argument "dt" but got 0.5 (unit is 1).
```

Numpy functions have been overwritten to correctly work with units (see the *developer documentation* for more details):

```
>>> print(mean(rates))
20. Hz
>>> print(rates.repeat(2))
[ 10.  10.  20.  20.  30.  30.] Hz
```

3.2.2 Removing units

There are various options to remove the units from a value (e.g. to use it with analysis functions that do not correctly work with units)

- Divide the value by its unit (most of the time the recommended option because it is clear about the scale)

- Transform it to a pure numpy array in the base unit by calling `asarray()` (no copy) or `array` (copy)

- Directly get the unitless value of a state variable by appending an underscore to the name

```
>>> tau/ms
20.0
>>> asarray(rates)
array([ 10.,  20.,  30.])
>>> G = NeuronGroup(5, 'dv/dt = -v/tau: volt')
>>> print(G.v_[:])
[ 0.  0.  0.  0.  0.]
```

3.2.3 Temperatures

Brian only supports temperatures defined in °K, using the provided `kelvin` unit object. Other conventions such as °C, or °F are not compatible with Brian's unit system, because they cannot be expressed as a multiplicative scaling of the SI base unit kelvin (their zero point is different). However, in biological experiments and modeling, temperatures are typically reported in °C. How to use such temperatures depends on whether they are used as *temperature differences* or as *absolute temperatures*:

temperature differences Their major use case is the correction of time constants for differences in temperatures based on the Q10 temperature coefficient. In this case, all temperatures can directly use `kelvin` even though the temperatures are reported in Celsius, since temperature differences in Celsius and Kelvin are identical.

absolute temperatures Equations such as the Goldman–Hodgkin–Katz voltage equation have a factor that depends on the absolute temperature measured in Kelvin. To get this temperature from a temperature reported in °C, you can use the `zero_celsius` constant from the *brian2.units.constants* package (see below):

```
from brian2.units.constants import zero_celsius

celsius_temp = 27
abs_temp = celsius_temp*kelvin + zero_celsius
```

Note: Earlier versions of Brian had a `celsius` unit which was in fact identical to `kelvin`. While this gave the correct results for temperature differences, it did not correctly work for absolute temperatures. To avoid confusion and possible misinterpretation, the `celsius` unit has therefore been removed.

3.2.4 Constants

The *brian2.units.constants* package provides a range of physical constants that can be useful for detailed biological models. Brian provides the following constants:

Constant	Symbol(s)	Brian name	Value
Avogadro constant	N_A, L	`avogadro_constant`	$6.022140857 \times 10^{23}\,\mathrm{mol}^{-1}$
Boltzmann constant	k	`boltzmann_constant`	$1.38064852 \times 10^{-23}\,\mathrm{J\,K}^{-1}$
Electric constant	ϵ_0	`electric_constant`	$8.854187817 \times 10^{-12}\,\mathrm{F\,m}^{-1}$
Electron mass	m_e	`electron_mass`	$9.10938356 \times 10^{-31}\,\mathrm{kg}$
Elementary charge	e	`elementary_charge`	$1.6021766208 \times 10^{-19}\,\mathrm{C}$
Faraday constant	F	`faraday_constant`	$96485.33289\,\mathrm{C\,mol}^{-1}$
Gas constant	R	`gas_constant`	$8.3144598\,\mathrm{J\,mol}^{-1}\,\mathrm{K}^{-1}$
Magnetic constant	μ_0	`magnetic_constant`	$12.566370614 \times 10^{-7}\,\mathrm{N\,A}^{-2}$
Molar mass constant	M_u	`molar_mass_constant`	$1 \times 10^{-3}\,\mathrm{kg\,mol}^{-1}$
0°C		`zero_celsius`	$273.15\,\mathrm{K}$

Note that these constants are not imported by default, you will have to explicitly import them from *brian2.units.constants*. During the import, you can also give them shorter names using Python's `from ... import ... as ...` syntax. For example, to calculate the $\frac{RT}{F}$ factor that appears in the Goldman–Hodgkin–Katz voltage equation you can use:

```
from brian2 import *
from brian2.units.constants import zero_celsius, gas_constant as R, faraday_constant↲
↪as F

celsius_temp = 27
T = celsius_temp*kelvin + zero_celsius
factor = R*T/F
```

The following topics are not essential for beginners.

3.2.5 Importing units

Brian generates standard names for units, combining the unit name (e.g. "siemens") with a prefixes (e.g. "m"), and also generates squared and cubed versions by appending a number. For example, the units "msiemens", "siemens2", "usiemens3" are all predefined. You can import these units from the package `brian2.units.allunits` – accordingly, an `from brian2.units.allunits import *` will result in everything from `Ylumen3` (cubed yotta lumen) to `ymol` (yocto mole) being imported.

A better choice is normally to do `from brian2.units import *` or import everything `from brian2 import *` which only imports the units mentioned in the introductory paragraph (base units, derived units, and some standard abbreviations).

3.2.6 In-place operations on quantities

In-place operations on quantity arrays change the underlying array, in the same way as for standard numpy arrays. This means, that any other variables referencing the same object will be affected as well:

```
>>> q = [1, 2] * mV
>>> r = q
>>> q += 1*mV
>>> q
array([ 2.,  3.]) * mvolt
>>> r
array([ 2.,  3.]) * mvolt
```

In contrast, scalar quantities will never change the underlying value but instead return a new value (in the same way as standard Python scalars):

```
>>> x = 1*mV
>>> y = x
>>> x *= 2
>>> x
2. * mvolt
>>> y
1. * mvolt
```

3.3 Models and neuron groups

For Brian 1 users

See the document *Neural models (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

- *Model equations*
- *Noise*
- *Threshold and reset*
- *Refractoriness*
- *State variables*

- *Subgroups*

- *Shared variables*

- *Storing state variables*

- *Linked variables*

- *Time scaling of noise*

3.3.1 Model equations

The core of every simulation is a `NeuronGroup`, a group of neurons that share the same equations defining their properties. The minimum `NeuronGroup` specification contains the number of neurons and the model description in the form of equations:

```
G = NeuronGroup(10, 'dv/dt = -v/(10*ms) : volt')
```

This defines a group of 10 leaky integrators. The model description can be directly given as a (possibly multi-line) string as above, or as an `Equations` object. For more details on the form of equations, see *Equations*. Brian needs the model to be given in the form of differential equations, but you might see the integrated form of synapses in some textbooks and papers. See *Converting from integrated form to ODEs* for details on how to convert between these representations.

Note that model descriptions can make reference to physical units, but also to scalar variables declared outside of the model description itself:

```
tau = 10*ms
G = NeuronGroup(10, 'dv/dt = -v/tau : volt')
```

If a variable should be taken as a *parameter* of the neurons, i.e. if it should be possible to vary its value across neurons, it has to be declared as part of the model description:

```
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                       tau : second''')
```

To make complex model descriptions more readable, named subexpressions can be used:

```
G = NeuronGroup(10, '''dv/dt = I_leak / Cm : volt
                       I_leak = g_L*(E_L - v) : amp''')
```

For a list of some standard model equations, see *Neural models (Brian 1 –> 2 conversion)*.

3.3.2 Noise

In addition to ordinary differential equations, Brian allows you to introduce random noise by specifying a stochastic differential equation. Brian uses the physicists' notation used in the Langevin equation, representing the "noise" as a term $\xi(t)$, rather than the mathematicians' stochastic differential dW_t. The following is an example of the Ornstein-Uhlenbeck process that is often used to model a leaky integrate-and-fire neuron with a stochastic current:

```
G = NeuronGroup(10, 'dv/dt = -v/tau + sigma*sqrt(2/tau)*xi : volt')
```

You can start by thinking of `xi` as just a Gaussian random variable with mean 0 and standard deviation 1. However, it scales in an unusual way with time and this gives it units of `1/sqrt(second)`. You don't necessarily need to understand why this is, but it is possible to get a reasonably simple intuition for it by thinking about numerical integration: *see below*.

Note: If you want to use noise in more than one equation of a `NeuronGroup` or `Synapses`, you will have to use suffixed names (see *Equation strings* for details).

3.3.3 Threshold and reset

To emit spikes, neurons need a *threshold*. Threshold and reset are given as strings in the `NeuronGroup` constructor:

```
tau = 10*ms
G = NeuronGroup(10, 'dv/dt = -v/tau : volt', threshold='v > -50*mV',
                reset='v = -70*mV')
```

Whenever the threshold condition is fulfilled, the reset statements will be executed. Again, both threshold and reset can refer to physical units, external variables and parameters, in the same way as model descriptions:

```
v_r = -70*mV  # reset potential
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                       v_th : volt  # neuron-specific threshold''',
                threshold='v > v_th', reset='v = v_r')
```

You can also create non-spike events. See *Custom events* for more details.

3.3.4 Refractoriness

To make a neuron non-excitable for a certain time period after a spike, the refractory keyword can be used:

```
G = NeuronGroup(10, 'dv/dt = -v/tau : volt', threshold='v > -50*mV',
                reset='v = -70*mV', refractory=5*ms)
```

This will not allow any threshold crossing for a neuron for 5ms after a spike. The refractory keyword allows for more flexible refractoriness specifications, see *Refractoriness* for details.

3.3.5 State variables

Differential equations and parameters in model descriptions are stored as *state variables* of the `NeuronGroup`. In addition to these variables, Brian also defines two variables automatically:

i The index of a neuron.

N The total number of neurons.

All state variables can be accessed and set as an attribute of the group. To get the values without physical units (e.g. for analysing data with external tools), use an underscore after the name:

```
>>> G = NeuronGroup(10, '''dv/dt = -v/tau : volt
...                         tau : second''', name='neurons')
>>> G.v = -70*mV
>>> G.v
<neurons.v: array([-70., -70., -70., -70., -70., -70., -70., -70., -70., -70.]) *␣
↪mvolt>
>>> G.v_  # values without units
<neurons.v_: array([-0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.07, -0.
↪07])>
```

The value of state variables can also be set using string expressions that can refer to units and external variables, other state variables or mathematical functions:

```
>>> G.tau = '5*ms + (1.0*i/N)*5*ms'
>>> G.tau
<neurons.tau: array([ 5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5]) *␣
↪msecond>
```

You can also set the value only if a condition holds, for example:

```
>>> G.v['tau>7.25*ms'] = -60*mV
>>> G.v
<neurons.v: array([-70., -70., -70., -70., -70., -60., -60., -60., -60., -60.]) *␣
↪mvolt>
```

3.3.6 Subgroups

It is often useful to refer to a subset of neurons, this can be achieved using Python's slicing syntax:

```
G = NeuronGroup(10, '''dv/dt = -v/tau : volt
                        tau : second''',
                threshold='v > -50*mV',
                reset='v = -70*mV')
# Create subgroups
G1 = G[:5]
G2 = G[5:]

# This will set the values in the main group, subgroups are just "views"
G1.tau = 10*ms
G2.tau = 20*ms
```

Here `G1` refers to the first 5 neurons in G, and `G2` to the second 5 neurons. In general `G[i:j]` refers to the neurons with indices from `i` to `j-1`, as in general in Python.

For convenience, you can also use a single index, i.e. `G[i]` is equivalent to `G[i:i+1]`. In some situations, it can be easier to provide a list of indices instead of a slice, Brian therefore also allows for this syntax. Note that this is restricted to cases that are strictly equivalent with slicing syntax, e.g. you can write `G[[3, 4, 5]]` instead of `G[3:6]`, but you *cannot* write `G[[3, 5, 7]]` or `G[[5, 4, 3]]`.

Subgroups can be used in most places where regular groups are used, e.g. their state variables or spiking activity can be recorded using monitors, they can be connected via *Synapses*, etc. In such situations, indices (e.g. the indices of the neurons to record from in a *StateMonitor*) are relative to the subgroup, not to the main group

The following topics are not essential for beginners.

3.3.7 Shared variables

Sometimes it can also be useful to introduce shared variables or subexpressions, i.e. variables that have a common value for all neurons. In contrast to external variables (such as `Cm` above), such variables can change during a run, e.g. by using `run_regularly()`. This can be for example used for an external stimulus that changes in the course of a run:

```
>>> G = NeuronGroup(10, '''shared_input : volt (shared)
...                        dv/dt = (-v + shared_input)/tau : volt
...                        tau : second''', name='neurons')
```

Note that there are several restrictions around the use of shared variables: they cannot be written to in contexts where statements apply only to a subset of neurons (e.g. reset statements, see below). If a code block mixes statements writing to shared and vector variables, then the shared statements have to come first.

By default, subexpressions are re-evaluated whenever they are used, i.e. using a subexpression is completely equivalent to substituting it. Sometimes it is useful to instead only evaluate a subexpression once and then use this value for the rest of the time step. This can be achieved by using the `(constant over dt)` flag. This flag is mandatory for subexpressions that refer to stateful functions like `rand()` which notably allows them to be recorded with a *StateMonitor* – otherwise the monitor would record a different instance of the random number than the one that was used in the equations.

For shared variables, setting by string expressions can only refer to shared values:

```
>>> G.shared_input = '(4.0/N)*mV'
>>> G.shared_input
<neurons.shared_input: 0.4 * mvolt>
```

3.3.8 Storing state variables

Sometimes it can be convenient to access multiple state variables at once, e.g. to set initial values from a dictionary of values or to store all the values of a group on disk. This can be done with the *get_states()* and *set_states()* methods:

```
>>> group = NeuronGroup(5, '''dv/dt = -v/tau : 1
...                           tau : second''', name='neurons')
>>> initial_values = {'v': [0, 1, 2, 3, 4],
...                   'tau': [10, 20, 10, 20, 10]*ms}
>>> group.set_states(initial_values)
>>> group.v[:]
array([ 0.,  1.,  2.,  3.,  4.])
>>> group.tau[:]
array([ 10.,  20.,  10.,  20.,  10.]) * msecond
>>> states = group.get_states()
>>> states['v']
array([ 0.,  1.,  2.,  3.,  4.])
```

The data (without physical units) can also be exported/imported to/from Pandas data frames (needs an installation of `pandas`):

```
>>> df = group.get_states(units=False, format='pandas')
>>> df
   N      dt  i    t   tau    v
0  5  0.0001  0  0.0  0.01  0.0
1  5  0.0001  1  0.0  0.02  1.0
2  5  0.0001  2  0.0  0.01  2.0
3  5  0.0001  3  0.0  0.02  3.0
4  5  0.0001  4  0.0  0.01  4.0
```

```
>>> df['tau']
0    0.01
1    0.02
2    0.01
3    0.02
4    0.01
Name: tau, dtype: float64
>>> df['tau'] *= 2
>>> group.set_states(df[['tau']], units=False, format='pandas')
>>> group.tau
<neurons.tau: array([ 20.,   40.,   20.,   40.,   20.]) * msecond>
```

3.3.9 Linked variables

A *NeuronGroup* can define parameters that are not stored in this group, but are instead a reference to a state variable in another group. For this, a group defines a parameter as linked and then uses *linked_var()* to specify the linking. This can for example be useful to model shared noise between cells:

```
inp = NeuronGroup(1, 'dnoise/dt = -noise/tau + tau**-0.5*xi : 1')

neurons = NeuronGroup(100, '''noise : 1 (linked)
                              dv/dt = (-v + noise_strength*noise)/tau : volt''')
neurons.noise = linked_var(inp, 'noise')
```

If the two groups have the same size, the linking will be done in a 1-to-1 fashion. If the source group has the size one (as in the above example) or if the source parameter is a shared variable, then the linking will be done as 1-to-all. In all other cases, you have to specify the indices to use for the linking explicitly:

```
# two inputs with different phases
inp = NeuronGroup(2, '''phase : 1
                        dx/dt = 1*mV/ms*sin(2*pi*100*Hz*t-phase) : volt''')
inp.phase = [0, pi/2]

neurons = NeuronGroup(100, '''inp : volt (linked)
                              dv/dt = (-v + inp) / tau : volt''')
# Half of the cells get the first input, other half gets the second
neurons.inp = linked_var(inp, 'x', index=repeat([0, 1], 50))
```

3.3.10 Time scaling of noise

Suppose we just had the differential equation

$$dx/dt = \xi$$

To solve this numerically, we could compute

$$x(t + \mathrm{d}t) = x(t) + \xi_1$$

where ξ_1 is a normally distributed random number with mean 0 and standard deviation 1. However, what happens if we change the time step? Suppose we used a value of $\mathrm{d}t/2$ instead of $\mathrm{d}t$. Now, we compute

$$x(t + \mathrm{d}t) = x(t + \mathrm{d}t/2) + \xi_1 = x(t) + \xi_2 + \xi_1$$

The mean value of $x(t + \mathrm{d}t)$ is 0 in both cases, but the standard deviations are different. The first method $x(t + \mathrm{d}t) = x(t) + \xi_1$ gives $x(t+\mathrm{d}t)$ a standard deviation of 1, whereas the second method $x(t+\mathrm{d}t) = x(t+\mathrm{d}/2)+\xi_1 = x(t)+\xi_2+\xi_1$

gives $x(t)$ a variance of 1+1=2 and therefore a standard deviation of $\sqrt{2}$.

In order to solve this problem, we use the rule $x(t + \mathrm{d}t) = x(t) + \sqrt{\mathrm{d}t}\xi_1$, which makes the mean and standard deviation of the value at time t independent of $\mathrm{d}t$. For this to make sense dimensionally, ξ must have units of `1/sqrt(second)`.

For further details, refer to a textbook on stochastic differential equations.

3.4 Numerical integration

By default, Brian chooses an integration method automatically, trying to solve the equations exactly first (for linear equations) and then resorting to numerical algorithms. It will also take care of integrating stochastic differential equations appropriately.

Note that in some cases, the automatic choice of integration method will not be appropriate, because of a choice of parameters that couldn't be determined in advance. In this case, typically you will get nan (not a number) values in the results, or large oscillations. In this case, Brian will generate a warning to let you know, but will not raise an error.

3.4.1 Method choice

You will get an `INFO` message telling you which integration method Brian decided to use, together with information about how much time it took to apply the integration method to your equations. If other methods have been tried but were not applicable, you will also see the time it took to try out those other methods. In some cases, checking other methods (in particular the `'exact'` method which attempts to solve the equations analytically) can take a considerable amount of time – to avoid wasting this time, you can always chose the integration method manually (see below). You can also suppress the message by raising the log level or by explicitly suppressing `'method_choice'` log messages – for details, see *Logging*.

If you prefer to chose an integration algorithm yourself, you can do so using the `method` keyword for *NeuronGroup*, *Synapses*, or *SpatialNeuron*. The complete list of available methods is the following:

- `'exact'`: exact integration for linear equations (alternative name: `'linear'`)
- `'exponential_euler'`: exponential Euler integration for conditionally linear equations
- `'euler'`: forward Euler integration (for additive stochastic differential equations using the Euler-Maruyama method)
- `'rk2'`: second order Runge-Kutta method (midpoint method)
- `'rk4'`: classical Runge-Kutta method (RK4)
- `'heun'`: stochastic Heun method for solving Stratonovich stochastic differential equations with non-diagonal multiplicative noise.
- `'milstein'`: derivative-free Milstein method for solving stochastic differential equations with diagonal multiplicative noise

---

Note: The `'independent'` integration method (exact integration for a system of independent equations, where all the equations can be analytically solved independently) should no longer be used and might be removed in future versions of Brian.

Note: The following methods are still considered experimental

---

- `'gsl'`: default integrator when choosing to integrate equations with the GNU Scientific Library ODE solver: the rkf45 method. Uses an adaptable time step by default.

- `'gsl_rkf45'`: Runge-Kutta-Fehlberg method. A good general-purpose integrator according to the GSL documentation. Uses an adaptable time step by default.

- `'gsl_rk2'`: Second order Runge-Kutta method using GSL. Uses an adaptable time step by default.

- `'gsl_rk4'`: Fourth order Runge-Kutta method using GSL. Uses an adaptable time step by default.

- `'gsl_rkck'`: Runge-Kutta Cash-Karp method using GSL. Uses an adaptable time step by default.

- `'gsl_rk8pd'`: Runge-Kutta Prince-Dormand method using GSL. Uses an adaptable time step by default.

The following topics are not essential for beginners.

3.4.2 Technical notes

Each class defines its own list of algorithms it tries to apply, *NeuronGroup* and *Synapses* will use the first suitable method out of the methods `'exact'`, `'euler'` and `'heun'` while *SpatialNeuron* objects will use `'exact'`, `'exponential_euler'`, `'rk2'` or `'heun'`.

You can also define your own numerical integrators, see *State update* for details.

3.4.3 GSL stateupdaters

The stateupdaters preceded with the gsl tag use ODE solvers defined in the GNU Scientific Library. The benefit of using these integrators over the ones written by Brian internally, is that they are implemented with an adaptable timestep. Integrating with an adaptable timestep comes with two advantages:

- These methods check whether the estimated error of the solutions returned fall within a certain error bound. For the non-gsl integrators there is currently no such check.

- Systems no longer need to be simulated with just one time step. That is, a bigger timestep can be chosen and the integrator will reduce the timestep when increased accuracy is required. This is particularly useful for systems where both slow and fast time constants coexist, as is the case with for example (networks of neurons with) Hodgkin-Huxley equations. Note that Brian's timestep still determines the resolution for monitors, spike timing, spike propagation etc. Hence, in a network, the simulation error will therefore still be on the order of `dt`. The benefit is that short time constants occurring in equations no longer dictate the network time step.

In addition to a choice between different integration methods, there are a few more options that can be specified when using GSL. These options can be specified by sending a dictionary as the `method_options` key upon initialization of the object using the integrator (*NeuronGroup*, *Synapses* or *SpatialNeuron*). The available method options are:

- `'adaptable_timestep'`: whether or not to let GSL reduce the timestep to achieve the accuracy defined with the `'absolute_error'` and `'absolute_error_per_variable'` options described below. If this is set to `False`, the timestep is determined by Brian (i.e. the `dt` of the respective clock is used, see *Scheduling*). If the resulted estimated error exceeds the set error bounds, the simulation is aborted. When using cython this is reported with an `IntegrationError`. Defaults to `True`.

- `'absolute_error'`: each of the methods has a way of estimating the error that is the result of using numerical integration. You can specify the maximum size of this error to be allowed for any of the to-be-integrated variables in base units with this keyword. Note that giving very small values makes the simulation slow and might result in unsuccessful integration. In the case of using the `'absolute_error_per_variable'` option, this is the error for variables that were not specified individually. Defaults to 1e-6.

- `'absolute_error_per_variable'`: specify the absolute error per variable in its own units. Variables for which the error is not specified use the error set with the `'absolute_error'` option. Defaults to None.

- `'max_steps'`: The maximal number of steps that the integrator will take within a single "Brian timestep" in order to reach the given error criterion. Can be set to 0 to not set any limits. Note that without limits, it can take a very long time until the integrator figures out that it cannot reach the desired error level. This will manifest as a simulation that appears to be stuck. Defaults to 100.

- `'use_last_timestep'`: with the `'adaptable_timestep'` option set to True, GSL tries different time steps to find a solution that satisfies the set error bounds. It is likely that for Brian's next time step the GSL time step will be somewhat similar per neuron (e.g. active neurons will have a shorter GSL time step than inactive neurons). With this option set to True, the time step GSL found to satisfy the set error bounds is saved per neuron and given to GSL again in Brian's next time step. This also means that the final time steps are saved in Brian's memory and can thus be recorded with the *StateMonitor*: it can be accessed under `'_last_timestep'`. Note that some extra memory is required to keep track of the last time steps. Defaults to True.

- `'save_failed_steps'`: if `'adaptable_timestep'` is set to True, each time GSL tries a time step and it results in an estimated error that exceeds the set bounds, one is added to the `'_failed_steps'` variable. For purposes of investigating what happens within GSL during an integration step, we offer the option of saving this variable. Defaults to False.

- `'save_step_count'`: the same goes for the total number of GSL steps taken in a single Brian time step: this is optionally saved in the `'_step_count'` variable. Defaults to False.

Note that at the moment recording `'_last_timestep'`, `'_failed_steps'`, or `'_step_count'` requires a call to *run()* (e.g. with 0 ms) to trigger the code generation process, before the call to *StateMonitor*.

More information on the GSL ODE solver itself can be found in its documentation.

3.5 Equations

- *Equation strings*
- *Arithmetic operations and functions*
- *External variables*
- *Flags*
- *List of special symbols*
- *Event-driven equations*
- *Equation objects*
- *Examples of* `Equation` *objects*

3.5.1 Equation strings

Equations are used both in *NeuronGroup* and *Synapses* to:

- define state variables

- define continuous-updates on these variables, through differential equations

---

Note: Brian models are defined by systems of first order ordinary differential equations, but you might see the integrated form of synapses in some textbooks and papers. See *Converting from integrated form to ODEs* for details on how to convert between these representations.

Equations are defined by multiline strings.

An Equation is a set of single lines in a string:

(1) `dx/dt = f : unit` (differential equation)

(2) `x = f : unit` (subexpression)

(3) `x : unit` (parameter)

Each equation may be spread out over multiple lines to improve formatting. Comments using # may also be included. Subunits are not allowed, i.e., one must write `volt`, not `mV`. This is to make it clear that the values are internally always saved in the basic units, so no confusion can arise when getting the values out of a *NeuronGroup* and discarding the units. Compound units are of course allowed as well (e.g. `farad/meter**2`). There are also three special "units" that can be used: `1` denotes a dimensionless floating point variable, `boolean` and `integer` denote dimensionless variables of the respective kind.

Note: For molar concentration, the base unit that has to be used in the equations is `mmolar` (or `mM`), *not* `molar`. This is because 1 molar is 10^3 mol/m^3 in SI units (i.e., it has a "scale" of 10^3), whereas 1 millimolar corresponds to 1 mol/m^3.

Some special variables are defined: `t`, `dt` (time) and `xi` (white noise). Variable names starting with an underscore and a couple of other names that have special meanings under certain circumstances (e.g. names ending in `_pre` or `_post`) are forbidden.

For stochastic equations with several `xi` values it is necessary to make clear whether they correspond to the same or different noise instantiations. To make this distinction, an arbitrary suffix can be used, e.g. using `xi_1` several times refers to the same variable, `xi_2` (or `xi_inh`, `xi_alpha`, etc.) refers to another. An error will be raised if you use more than one plain `xi`. Note that noise is always independent across neurons, you can only work around this restriction by defining your noise variable as a shared parameter and update it using a user-defined function (e.g. with `run_regularly`), or create a group that models the noise and link to its variable (see *Linked variables*).

3.5.2 Arithmetic operations and functions

Equation strings can make use of standard arithmetic operations for numerical values, using the Python 3 syntax. The supported operations are +, −, *, / (floating point division), // (flooring division), % (remainder), ** (power). For variable assignments, e.g. in reset statements, the corresponding in-place assignments such as += can be used as well. For comparisons, the operations == (equality), != (inequality), <, <=, >, and >= are available. Truth values can be combined using `and` and `or`, or negated using `not`. Note that Brian does not support any operations specific to integers, e.g. "bitwise AND" or shift operations.

> **Warning:** Brian versions up to 2.1.3.1 did not support `//` as the floor division operator and potentially used different semantics for the `/` operator depending on whether Python 2 or 3 was used. To write code that correctly and unambiguously works with both newer and older Brian versions, you can use expressions such as `1.0*a/b` to enforce floating point division (if one of the operands is a floating point number, both Python 2 and 3 will use floating point division), or `floor(a/b)` to enforce flooring division Note that the `floor` function always returns a floating point value, if it is important that the result is an integer value, additionally wrap it with the `int` function.

Brian also supports standard mathematical functions with the same names as used in the `numpy` library (e.g. `exp`, `sqrt`, `abs`, `clip`, `sin`, `cos`, …) – for a full list see *Default functions*. Note that support for such functions is provided by Brian itself and the translation to the various code generation targets is automatically taken care of. You should therefore refer to them directly by name and not as e.g. `np.sqrt` or `numpy.sqrt`, regardless of the way you *imported Brian or numpy*. This also means that you cannot directly refer to arbitrary functions from `numpy` or other libraries. For details on how to extend the support to non-default functions see *User-provided functions*.

3.5.3 External variables

Equations defining neuronal or synaptic equations can contain references to external parameters or functions. These references are looked up at the time that the simulation is run. If you don't specify where to look them up, it will look in the Python local/global namespace (i.e. the block of code where you call `run()`). If you want to override this, you can specify an explicit "namespace". This is a Python dictionary with keys being variable names as they appear in the equations, and values being the desired value of that variable. This namespace can be specified either in the creation of the group or when you can the `run()` function using the `namespace` keyword argument.

The following three examples show the different ways of providing external variable values, all having the same effect in this case:

```python
# Explicit argument to the NeuronGroup
G = NeuronGroup(1, 'dv/dt = -v / tau : 1', namespace={'tau': 10*ms})
net = Network(G)
net.run(10*ms)

# Explicit argument to the run function
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
net.run(10*ms, namespace={'tau': 10*ms})

# Implicit namespace from the context
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
tau = 10*ms
net.run(10*ms)
```

See *Namespaces* for more details.

The following topics are not essential for beginners.

3.5.4 Flags

A *flag* is a keyword in parentheses at the end of the line, which qualifies the equations. There are several keywords:

event-driven this is only used in Synapses, and means that the differential equation should be updated only at the times of events. This implies that the equation is taken out of the continuous state update, and instead a event-based state update statement is generated and inserted into event codes (pre and post). This can only qualify differential equations of synapses. Currently, only one-dimensional linear equations can be handled (see below).

unless refractory this means the variable is not updated during the refractory period. This can only qualify differential equations of neuron groups.

constant this means the parameter will not be changed during a run. This allows optimizations in state updaters. This can only qualify parameters.

constant over dt this means that the subexpression will be only evaluated once at the beginning of the time step. This can be useful to e.g. approximate a non-linear term as constant over a time step in order to use the `linear` numerical integration algorithm. It is also mandatory for subexpressions that refer to stateful functions like `rand()` to make sure that they are only evaluated once (otherwise e.g. recording the value with a *StateMonitor* would re-evaluate it and therefore not record the same values that are used in other places). This can only qualify subexpressions.

shared this means that a parameter or subexpression is not neuron-/synapse-specific but rather a single value for the whole *NeuronGroup* or *Synapses*. A shared subexpression can only refer to other shared variables.

linked this means that a parameter refers to a parameter in another *NeuronGroup*. See *Linked variables* for more details.

Multiple flags may be specified as follows:

```
dx/dt = f : unit (flag1,flag2)
```

3.5.5 List of special symbols

The following lists all of the special symbols that Brian uses in equations and code blocks, and their meanings.

dt Time step width

i Index of a neuron (*NeuronGroup*) or the pre-synaptic neuron of a synapse (*Synapses*)

j Index of a post-synaptic neuron of a synapse

lastspike Last time that the neuron spiked (for refractoriness)

lastupdate Time of the last update of synaptic variables in event-driven equations (only defined when event-driven equations are used).

N Number of neurons (*NeuronGroup*) or synapses (*Synapses*). Use `N_pre` or `N_post` for the number of presynaptic or postsynaptic neurons in the context of *Synapses*.

not_refractory Boolean variable that is normally true, and false if the neuron is currently in a refractory state

t Current time

t_in_timesteps Current time measured in time steps

xi, xi_* Stochastic differential in equations

3.5.6 Event-driven equations

Equations defined as event-driven are completely ignored in the state update. They are only defined as variables that can be externally accessed. There are additional constraints:

- An event-driven variable cannot be used by any other equation that is not also event-driven.

- An event-driven equation cannot depend on a differential equation that is not event-driven (directly, or indirectly through subexpressions). It can depend on a constant parameter.

Currently, automatic event-driven updates are only possible for one-dimensional linear equations, but this may be extended in the future.

3.5.7 Equation objects

The model definitions for *NeuronGroup* and *Synapses* can be simple strings or *Equations* objects. Such objects can be combined using the add operator:

```
eqs = Equations('dx/dt = (y-x)/tau : volt')
eqs += Equations('dy/dt = -y/tau: volt')
```

Equations allow for the specification of values in the strings, but does this by simple string replacement, e.g. you can do:

```
eqs = Equations('dx/dt = x/tau : volt', tau=10*ms)
```

but this is exactly equivalent to:

```
eqs = Equations('dx/dt = x/(10*ms) : volt')
```

The *Equations* object does some basic syntax checking and will raise an error if two equations defining the same variable are combined. It does not however do unit checking, checking for unknown identifiers or incorrect flags – all this will be done during the instantiation of a *NeuronGroup* or *Synapses* object.

3.5.8 Examples of `Equation` objects

Concatenating equations

```
>>> membrane_eqs = Equations('dv/dt = -(v + I)/ tau : volt')
>>> eqs1 = membrane_eqs + Equations('''I = sin(2*pi*freq*t) : volt
...                                     freq : Hz''')
>>> eqs2 = membrane_eqs + Equations('''I : volt''')
>>> print(eqs1)
I = sin(2*pi*freq*t) : V
dv/dt = -(v + I)/ tau : V
freq : Hz
>>> print(eqs2)
dv/dt = -(v + I)/ tau : V
I : V
```

Substituting variable names

```
>>> general_equation = 'dg/dt = -g / tau : siemens'
>>> eqs_exc = Equations(general_equation, g='g_e', tau='tau_e')
>>> eqs_inh = Equations(general_equation, g='g_i', tau='tau_i')
```

```
>>> print(eqs_exc)
dg_e/dt = -g_e / tau_e : S
>>> print(eqs_inh)
dg_i/dt = -g_i / tau_i : S
```

Inserting values

```
>>> eqs = Equations('dv/dt = mu/tau + sigma/tau**.5*xi : volt',
...                  mu=-65*mV, sigma=3*mV, tau=10*ms)
>>> print(eqs)
dv/dt = (-65. * mvolt)/(10. * msecond) + (3. * mvolt)/(10. * msecond)**.5*xi : V
```

3.6 Refractoriness

- *Defining the refractory period*
- *Defining model behaviour during refractoriness*
- *Arbitrary refractoriness*

Brian allows you to model the absolute refractory period of a neuron in a flexible way. The definition of refractoriness consists of two components: the amount of time after a spike that a neuron is considered to be refractory, and what changes in the neuron during the refractoriness.

3.6.1 Defining the refractory period

The refractory period is specified by the `refractory` keyword in the *NeuronGroup* initializer. In the simplest case, this is simply a fixed time, valid for all neurons:

```
G = NeuronGroup(N, model='...', threshold='...', reset='...',
                refractory=2*ms)
```

Alternatively, it can be a string expression that evaluates to a time. This expression will be evaluated after every spike and allows for a varying refractory period. For example, the following will set the refractory period to a random duration between 1ms and 3ms after every spike:

```
G = NeuronGroup(N, model='...', threshold='...', reset='...',
                refractory='(1 + 2*rand())*ms')
```

In general, modelling a refractory period that varies across neurons involves declaring a state variable that stores the refractory period per neuron as a model parameter. The refractory expression can then refer to this parameter:

```
G = NeuronGroup(N, model='''...
                           refractory : second''', threshold='...',
                reset='...', refractory='refractory')
# Set the refractory period for each cell
G.refractory = ...
```

This state variable can also be a dynamic variable itself. For example, it can serve as an adaptation mechanism by increasing it after every spike and letting it relax back to a steady-state value between spikes:

```
refractory_0 = 2*ms
tau_refractory = 50*ms
G = NeuronGroup(N, model='''...
                        drefractory/dt = (refractory_0 - refractory) / tau_
↪refractory : second''',
              threshold='...', refractory='refractory',
              reset='''...
                      refractory += 1*ms''')
G.refractory = refractory_0
```

In some cases, the condition for leaving the refractory period is not easily expressed as a certain time span. For example, in a Hodgkin-Huxley type model the threshold is only used for *counting* spikes and the refractoriness is used to prevent the count of multiple spikes for a single threshold crossing (the threshold condition would evaluate to `True` for several time points). When a neuron should leave the refractory period is not easily expressed as a time span but more naturally as a condition that the neuron should remain refractory for as long as it stays above the threshold. This can be achieved by using a string expression for the `refractory` keyword that evaluates to a boolean condition:

```
G = NeuronGroup(N, model='...', threshold='v > -20*mV',
              refractory='v >= -20*mV')
```

The `refractory` keyword should be read as "stay refractory as long as the condition remains true". In fact, specifying a time span for the refractoriness will be automatically transformed into a logical expression using the current time `t` and the time of the last spike `lastspike`. Specifying `refractory=2*ms` is basically equivalent to specifying `refractory='(t - lastspike) <= 2*ms'`. However, this expression can give inconsistent results for the common case that the refractory period is a multiple of the simulation timestep. Due to floating point impreciseness, the actual value of `t - lastspike` can be slightly above or below a multiple of the simulation time step; comparing it directly to the refractory period can therefore lead to an end of the refractory one time step sooner or later. To avoid this issue, the actual code used for the above example is equivalent to `refractory='timestep(t - lastspike, dt) <= timestep(2*ms, dt)'`. The *timestep* function is provided by Brian and takes care of converting a time into a time step in a safe way.

New in version 2.1.3: The `timestep` function is now used to avoid floating point issues in the refractoriness calculation. To restore the previous behaviour, set the *legacy.refractory_timing* preference to `True`.

3.6.2 Defining model behaviour during refractoriness

The refractoriness definition as described above only has a single effect by itself: threshold crossings during the refractory period are ignored. In the following model, the variable `v` continues to update during the refractory period but it does not elicit a spike if it crosses the threshold:

```
G = NeuronGroup(N, 'dv/dt = -v / tau : 1',
              threshold='v > 1', reset='v=0',
              refractory=2*ms)
```

There is also a second implementation of refractoriness that is supported by Brian, one or several state variables can be clamped during the refractory period. To model this kind of behaviour, variables that should stop being updated during refractoriness can be marked with the `(unless refractory)` flag:

```
G = NeuronGroup(N, '''dv/dt = -(v + w)/ tau_v : 1 (unless refractory)
                    dw/dt = -w / tau_w : 1''',
              threshold='v > 1', reset='v=0; w+=0.1', refractory=2*ms)
```

In the above model, the `v` variable is clamped at 0 for 2ms after a spike but the adaptation variable `w` continues to update during this time. In addition, a variable of a neuron that is in its refractory period is *read-only*: incoming synapses or other code will have no effect on the value of `v` until it leaves its refractory period.

The following topics are not essential for beginners.

3.6.3 Arbitrary refractoriness

In fact, arbitrary behaviours can be defined using Brian's refractoriness mechanism.

A *NeuronGroup* with refractoriness automatically defines two variables:

not_refractory A boolean variable stating whether a neuron is allowed to spike.

lastspike The time of the last spike of the neuron.

The variable `not_refractory` is updated at every time step by checking the refractoriness condition – for a refractoriness defined by a time period, this means comparing `lastspike` to the current time `t`. The `not_refractory` variable is then used to implement the refractoriness behaviour. Specifically, the `threshold` condition is replaced by `threshold and not_refractory` and differential equations that are marked as (`unless refractory`) are multiplied by `int(not_refractory)` (so that they have the value 0 when the neuron is refractory).

This `not_refractory` variable is also available to the user to define more sophisticated refractoriness behaviour. For example, the following code updates the `w` variable with a different time constant during refractoriness:

```
G = NeuronGroup(N, '''dv/dt = -(v + w)/ tau_v : 1 (unless refractory)
                      dw/dt = (-w / tau_active)*int(not_refractory) + (-w / tau_
→ref)*(1 - int(not_refractory)) : 1''',
                threshold='v > 1', reset='v=0; w+=0.1', refractory=2*ms)
```

3.7 Synapses

> **For Brian 1 users**
>
> *Synapses* is now the only class for defining synaptic interactions, it replaces *Connection*, *STDP*, etc. See the document *Synapses (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

- *Defining synaptic models*
- *Creating synapses*
- *Accessing synaptic variables*
- *Delays*
- *Monitoring synaptic variables*
- *Synaptic connection/weight matrices*
- *Creating synapses with the generator syntax*
- *Summed variables*

- *Creating multi-synapses*
- *Multiple pathways*
- *Numerical integration*
- *Technical notes*

3.7.1 Defining synaptic models

The most simple synapse (adding a fixed amount to the target membrane potential on every spike) is described as follows:

```
w = 1*mV
S = Synapses(P, Q, on_pre='v += w')
```

This defines a set of synapses between `NeuronGroup` P and `NeuronGroup` Q. If the target group is not specified, it is identical to the source group by default. The `on_pre` keyword defines what happens when a presynaptic spike arrives at a synapse. In this case, the constant `w` is added to variable `v`. Because `v` is not defined as a synaptic variable, it is assumed by default that it is a postsynaptic variable, defined in the target `NeuronGroup` Q. Note that this does not create synapses (see *Creating Synapses*), only the synaptic models.

To define more complex models, models can be described as string equations, similar to the models specified in *NeuronGroup*:

```
S = Synapses(P, Q, model='w : volt', on_pre='v += w')
```

The above specifies a parameter `w`, i.e. a synapse-specific weight. Note that to avoid confusion, synaptic variables cannot have the same name as a pre- or post-synaptic variables.

Synapses can also specify code that should be executed whenever a postsynaptic spike occurs (keyword `on_post`) and a fixed (pre-synaptic) delay for all synapses (keyword `delay`).

As shown above, variable names that are not referring to a synaptic variable are automatically understood to be post-synaptic variables. To explicitly specify that a variable should be from a pre- or post-synaptic neuron, append the suffix `_pre` or `_post`. An alternative but equivalent formulation of the `on_pre` statement above would therefore be `v_post += w`.

Model syntax

The model follows exactly the same syntax as for `NeuronGroup`. There can be parameters (e.g. synaptic variable `w` above), but there can also be named subexpressions and differential equations, describing the dynamics of synaptic variables. In all cases, synaptic variables are created, one value per synapse.

Brian also automatically defines a number of synaptic variables that can be used in equations, `on_pre` and `on_post` statements, as well as when *assigning to other synaptic variables*:

i The index of the pre-synaptic source of a synapse.

j The index of the post-synaptic target of a synapse.

N The total number of synapses.

N_incoming The total number of synapses connected to the post-synaptic target of a synapse.

N_outgoing The total number of synapses outgoing from the pre-synaptic source of a synapse.

lastupdate The last time this synapse has applied an `on_pre` or `on_post` statement. There is normally no need to refer to this variable explicitly, it is used to implement *Event-driven updates* (see below). It is only defined when event-driven equations are used.

Event-driven updates

By default, differential equations are integrated in a clock-driven fashion, as for a *NeuronGroup*. This is potentially very time consuming, because all synapses are updated at every timestep and Brian will therefore emit a warning. If you are sure about integrating the equations at every timestep (e.g. because you want to record the values continuously), then you should specify the flag `(clock-driven)`, which will silence the warning. To ask Brian 2 to simulate differential equations in an event-driven fashion use the flag `(event-driven)`. A typical example is pre- and postsynaptic traces in STDP:

```
model='''w:1
         dApre/dt=-Apre/taupre : 1 (event-driven)
         dApost/dt=-Apost/taupost : 1 (event-driven)'''
```

Here, Brian updates the value of `Apre` for a given synapse only when this synapse receives a spike, whether it is presynaptic or postsynaptic. More precisely, the variables are updated every time either the `on_pre` or `on_post` code is called for the synapse, so that the values are always up to date when these codes are executed.

Automatic event-driven updates are only possible for a subset of equations, in particular for one-dimensional linear equations. These equations must also be independent of the other ones, that is, a differential equation that is not event-driven cannot depend on an event-driven equation (since the values are not continuously updated). In other cases, the user can write event-driven code explicitly in the update codes (see below).

Pre and post codes

The `on_pre` code is executed at each synapse receiving a presynaptic spike. For example:

```
on_pre='v+=w'
```

adds the value of synaptic variable `w` to postsynaptic variable `v`. Any sort of code can be executed. For example, the following code defines stochastic synapses, with a synaptic weight `w` and transmission probability `p`:

```
S=Synapses(neuron_input,neurons,model="""w : 1
                     p : 1""",
                 on_pre="v+=w*(rand()<p)")
```

The code means that `w` is added to `v` with probability `p`. The code may also include multiple lines.

Similarly, the `on_post` code is executed at each synapse where the postsynaptic neuron has fired a spike.

3.7.2 Creating synapses

Creating a *Synapses* instance does not create synapses, it only specifies their dynamics. The following command creates a synapse between neuron `5` in the source group and neuron `10` in the target group:

```
S.connect(i=5, j=10)
```

Multiple synaptic connections can be created in a single statement:

```
S.connect()
S.connect(i=[1, 2], j=[3, 4])
S.connect(i=numpy.arange(10), j=1)
```

The first statement connects all neuron pairs. The second statement creates synapses between neurons 1 and 3, and between neurons 2 and 4. The third statement creates synapses between the first ten neurons in the source group and neuron 1 in the target group.

Conditional

One can also create synapses by giving (as a string) the condition for a pair of neurons i and j to be connected by a synapse, e.g. you could connect neurons that are not very far apart with:

```
S.connect(condition='abs(i-j)<=5')
```

The string expressions can also refer to pre- or postsynaptic variables. This can be useful for example for spatial connectivity: assuming that the pre- and postsynaptic groups have parameters x and y, storing their location, the following statement connects all cells in a 250 um radius:

```
S.connect(condition='sqrt((x_pre-x_post)**2 + (y_pre-y_post)**2) < 250*umeter')
```

Probabilistic

Synapse creation can also be probabilistic by providing a p argument, providing the connection probability for each pair of synapses:

```
S.connect(p=0.1)
```

This connects all neuron pairs with a probability of 10%. Probabilities can also be given as expressions, for example to implement a connection probability that depends on distance:

```
S.connect(condition='i != j',
          p='p_max*exp(-(x_pre-x_post)**2+(y_pre-y_post)**2 / (2*(125*umeter)**2))')
```

If this statement is applied to a *Synapses* object that connects a group to itself, it prevents self-connections (i != j) and connects cells with a probability that is modulated according to a 2-dimensional Gaussian of the distance between the cells.

One-to-one

You can specify a mapping from i to any function f(i), e.g. the simplest way to give a 1-to-1 connection would be:

```
S.connect(j='i')
```

This mapping can also use a restricting condition with if, e.g. to connect neurons 0, 2, 4, 6, … to neurons 0, 1, 2, 3, … you could write:

```
S.connect(j='int(i/2) if i % 2 == 0')
```

The connections above describe the target indices j as a function of the source indices i. You can also apply the syntax in the other direction, i.e. describe source indices i as a function of target indices j. For a 1-to-1 connection, this does not change anything in most cases:

```
S.connect(i='j')
```

Note that there is a subtle difference between the two descriptions if the two groups do not have the same size: if the source group has fewer neurons than the target group, then using j='i' is possible (there is a target neuron for each source neuron), but i='j' would raise an error; the opposite is true if the source group is bigger than the target group.

The second example from above (neurons 0, 2, 4, … to neurons 0, 1, 2, …) can be adapted for the other direction, as well, and is possibly more intuitive in this case:

```
S.connect(i='j*2')
```

3.7.3 Accessing synaptic variables

Synaptic variables can be accessed in a similar way as `NeuronGroup` variables. They can be indexed with two indexes, corresponding to the indexes of pre and postsynaptic neurons, or with string expressions (referring to `i` and `j` as the pre-/post-synaptic indices, or to other state variables of the synapse or the connected neurons). Note that setting a synaptic variable always refers to the synapses that *currently exist*, i.e. you have to set them *after* the relevant `Synapses.connect` call.

Here are a few examples:

```
S.w[2, 5] = 1*nS
S.w[1, :] = 2*nS
S.w = 1*nS # all synapses assigned
S.w[2, 3] = (1*nS, 2*nS)
S.w[group1, group2] = "(1+cos(i-j))*2*nS"
S.w[:, :] = 'rand()*nS'
S.w['abs(x_pre-x_post) < 250*umetre'] = 1*nS
```

Assignments can also refer to *pre-defined variables*, e.g. to normalize synaptic weights. For example, after the following assignment the sum of weights of all synapses that a neuron receives is identical to 1, regardless of the number of synapses it receives:

```
syn.w = '1.0/N_incoming'
```

Note that it is also possible to index synaptic variables with a single index (integer, slice, or array), but in this case synaptic indices have to be provided.

The `N_incoming` and `N_outgoing` variables give access to the total number of incoming/outgoing synapses for a neuron, but this access is given for each *synapse*. This is necessary to apply it to individual synapses as in the statement to normalize synaptic weights mentioned above. To access these values per *neuron* instead, `N_incoming_post` and `N_outgoing_pre` can be used. Note that synaptic equations or `on_pre`/`on_post` statements should always refer to `N_incoming` and `N_outgoing` without `pre`/`post` suffix.

Here's a little example illustrating the use of these variables:

```
>>> group1 = NeuronGroup(3, '')
>>> group2 = NeuronGroup(3, '')
>>> syn = Synapses(group1, group2)
>>> syn.connect(i=[0, 0, 1, 2], j=[1, 2, 2, 2])
>>> print(syn.N_outgoing_pre)  # for each presynaptic neuron
[2 1 1]
>>> print(syn.N_outgoing[:])   # same numbers, but indexed by synapse
[2 2 1 1]
>>> print(syn.N_incoming_post)
[0 1 3]
>>> print(syn.N_incoming[:])
[1 3 3 3]
```

Note that `N_incoming_post` and `N_outgoing_pre` can contain zeros for neurons that do not have any incoming respectively outgoing synapses. In contrast, `N_incoming` and `N_outgoing` will never contain zeros, because unconnected neurons are not represented in the list of synapses.

3.7.4 Delays

There is a special synaptic variable that is automatically created: `delay`. It is the propagation delay from the presynaptic neuron to the synapse, i.e., the presynaptic delay. This is just a convenience syntax for accessing the delay stored in the presynaptic pathway: `pre.delay`. When there is a postsynaptic code (keyword `post`), the delay of the postsynaptic pathway can be accessed as `post.delay`.

The delay variable(s) can be set and accessed in the same way as other synaptic variables. The same semantics as for other synaptic variables apply, which means in particular that the delay is only set for the synapses that have been already created with `Synapses.connect`. If you want to set a global delay for all synapses of a `Synapses` object, you can directly specify that delay as part of the `Synapses` initializer:

```
synapses = Synapses(sources, targets, '...', on_pre='...', delay=1*ms)
```

When you use this syntax, you can still change the delay afterwards by setting `synapses.delay`, but you can only set it to another scalar value. If you need different delays across synapses, do not use this syntax but instead set the delay variable as any other synaptic variable (see above).

3.7.5 Monitoring synaptic variables

A `StateMonitor` object can be used to monitor synaptic variables. For example, the following statement creates a monitor for variable `w` for the synapses 0 and 1:

```
M = StateMonitor(S, 'w', record=[0,1])
```

Note that these are *synapse* indices, not neuron indices. More convenient is to directly index the `Synapses` object, Brian will automatically calculate the indices for you in this case:

```
M = StateMonitor(S, 'w', record=S[0, :])  # all synapses originating from neuron 0
M = StateMonitor(S, 'w', record=S['i!=j'])  # all synapses excluding autapses
M = StateMonitor(S, 'w', record=S['w>0'])  # all synapses with non-zero weights (at
↪this time)
```

You can also record a synaptic variable for all synapses by passing `record=True`.

The recorded traces can then be accessed in the usual way, again with the possibility to index the `Synapses` object:

```
plot(M.t / ms, M[S[0]].w / nS)  # first synapse
plot(M.t / ms, M[S[0, :]].w / nS)  # all synapses originating from neuron 0
plot(M.t / ms, M[S['w>0*nS']].w / nS)  # all synapses with non-zero weights (at this
↪time)
```

Note (for users of Brian's advanced standalone mode only): the use of the `Synapses` object for indexing and `record=True` only work in the default runtime modes. In standalone mode (see *Standalone code generation*), the synapses have not yet been created at this point, so Brian cannot calculate the indices.

The following topics are not essential for beginners.

3.7.6 Synaptic connection/weight matrices

Brian does not directly support specifying synapses by using a matrix, you always have to use a "sparse" format, where each connection is defined by its source and target indices. However, you can easily convert between the two formats. Assuming you have a connection matrix C of size $N \times M$, where N is the number of presynaptic cells, and M the number of postsynaptic cells, with each entry being 1 for a connection, and 0 otherwise. You can convert this matrix to arrays of source and target indices, which you can then provide to Brian's `connect` function:

```
C = ...   # The connection matrix as a numpy array of 0's and 1's
sources, targets = C.nonzero()
synapses = Synapses(...)
synapses.connect(i=sources, j=targets)
```

Similarly, you can transform the flat array of values stored in a synapse into a matrix form. For example, to get a matrix with all the weight values `w`, with `NaN` values where no synapse exists:

```
synapses = Synapses(source_group, target_group,
                    '''...
                       w : 1  # synaptic weight''', ...)
# ...
# Run e.g. a simulation with plasticity that changes the weights
run(...)
# Create a matrix to store the weights and fill it with NaN
W = np.full((len(source_group), len(target_group)), np.nan)
# Insert the values from the Synapses object
W[synapses.i[:], synapses.j[:]] = synapses.w[:]
```

3.7.7 Creating synapses with the generator syntax

The most general way of specifying a connection is using the generator syntax, e.g. to connect neuron i to all neurons j with 0<=j<=i:

```
S.connect(j='k for k in range(0, i+1)')
```

There are several parts to this syntax. The general form is:

```
j='EXPR for VAR in RANGE if COND'
```

or:

```
i='EXPR for VAR in RANGE if COND'
```

Here `EXPR` can be any integer-valued expression. VAR is the name of the iteration variable (any name you like can be specified here). The `if COND` part is optional and lets you give an additional condition that has to be true for the synapse to be created. Finally, `RANGE` can be either:

1. a Python `range`, e.g. `range(N)` is the integers from 0 to N-1, `range(A, B)` is the integers from A to B-1, `range(low, high, step)` is the integers from `low` to `high-1` with steps of size `step`;

2. a random sample `sample(N, p=0.1)` gives a random sample of integers from 0 to N-1 with 10% probability of each integer appearing in the sample. This can have extra arguments like range, e.g. `sample(low, high, step, p=0.1)` will give each integer in `range(low, high, step)` with probability 10%;

3. a random sample `sample(N, size=10)` with a fixed size, in this example 10 values chosen (without replacement) from the integers from 0 to N-1. As for the random sample based on a probability, the `sample` expression can take additional arguments to sample from a restricted range.

If you try to create an invalid synapse (i.e. connecting neurons that are outside the correct range) then you will get an error, e.g. you might like to try to do this to connect each neuron to its neighbours:

```
S.connect(j='i+(-1)**k for k in range(2)')
```

However this won't work at for `i=0` it gives `j=-1` which is invalid. There is an option to just skip any synapses that are outside the valid range:

```
S.connect(j='i+(-1)**k for k in range(2)', skip_if_invalid=True)
```

You can also use this argument to deal with random samples of incorrect size, i.e. a negative size or a size bigger than the total population size. With `skip_if_invalid=True`, no error will be raised and a size of 0 or the population size will be used.

3.7.8 Summed variables

In many cases, the postsynaptic neuron has a variable that represents a sum of variables over all its synapses. This is called a "summed variable". An example is nonlinear synapses (e.g. NMDA):

```
neurons = NeuronGroup(1, model='''dv/dt=(gtot-v)/(10*ms) : 1
                                   gtot : 1''')
S = Synapses(neuron_input, neurons,
            model='''dg/dt=-a*g+b*x*(1-g) : 1
                     gtot_post = g : 1   (summed)
                     dx/dt=-c*x : 1
                     w : 1 # synaptic weight''', on_pre='x+=w')
```

Here, each synapse has a conductance `g` with nonlinear dynamics. The neuron's total conductance is `gtot`. The line stating `gtot_post = g : 1 (summed)` specifies the link between the two: `gtot` in the postsynaptic group is the summer over all variables `g` of the corresponding synapses. What happens during the simulation is that at each time step, presynaptic conductances are summed for each neuron and the result is copied to the variable `gtot`. Another example is gap junctions:

```
neurons = NeuronGroup(N, model='''dv/dt=(v0-v+Igap)/tau : 1
                                  Igap : 1''')
S=Synapses(neurons,model='''w:1 # gap junction conductance
                            Igap_post = w*(v_pre-v_post): 1 (summed)''')
```

Here, `Igap` is the total gap junction current received by the postsynaptic neuron.

Note that you cannot target the same post-synaptic variable from more than one *Synapses* object. To work around this restriction, use multiple post-synaptic variables that ar then summed up:

```
neurons = NeuronGroup(1, model='''dv/dt=(gtot-v)/(10*ms) : 1
                                  gtot = gtot1 + gtot2: 1
                                  gtot1 : 1
                                  gtot2 : 1''')
S1 = Synapses(neuron_input, neurons,
            model='''dg/dt=-a1*g+b1*x*(1-g) : 1
                     gtot1_post = g : 1   (summed)
                     dx/dt=-c1*x : 1
                     w : 1 # synaptic weight
                 ''', on_pre='x+=w')
S2 = Synapses(neuron_input, neurons,
            model='''dg/dt=-a2*g+b2*x*(1-g) : 1
                     gtot2_post = g : 1   (summed)
```

```
                    dx/dt=-c2*x : 1
                    w : 1 # synaptic weight
               ''', on_pre='x+=w')
```

3.7.9 Creating multi-synapses

It is also possible to create several synapses for a given pair of neurons:

```
S.connect(i=numpy.arange(10), j=1, n=3)
```

This is useful for example if one wants to have multiple synapses with different delays. To distinguish multiple variables connecting the same pair of neurons in synaptic expressions and statements, you can create a variable storing the synapse index with the `multisynaptic_index` keyword:

```
syn = Synapses(source_group, target_group, model='w : 1', on_pre='v += w',
               multisynaptic_index='synapse_number')
syn.connect(i=numpy.arange(10), j=1, n=3)
syn.delay = '1*ms + synapse_number*2*ms'
```

This index can then be used to set/get synapse-specific values:

```
S.delay = '(synapse_number + 1)*ms)'   # Set delays between 1 and 10ms
S.w['synapse_number<5'] = 0.5
S.w['synapse_number>=5'] = 1
```

It also enables three-dimensional indexing, the following statement has the same effect as the last one above:

```
S.w[:, :, 5:] = 1
```

3.7.10 Multiple pathways

It is possible to have multiple pathways with different update codes from the same presynaptic neuron group. This may be interesting in cases when different operations must be applied at different times for the same presynaptic spike, e.g. for a STDP rule that shifted in time. To do this, specify a dictionary of pathway names and codes:

```
on_pre={'pre_transmission': 'ge+=w',
        'pre_plasticity': '''w=clip(w+Apost,0,inf)
                             Apre+=dApre'''}
```

This creates two pathways with the given names (in fact, specifying `on_pre=code` is just a shorter syntax for `on_pre={'pre':  code}`) through which the delay variables can be accessed. The following statement, for example, sets the delay of the synapse between the first neurons of the source and target groups in the `pre_plasticity` pathway:

```
S.pre_plasticity.delay[0,0] = 3*ms
```

As mentioned above, `pre` pathways are generally executed before `post` pathways. The order of execution of several `pre` (or `post`) pathways with the same delay is however arbitrary, and simply based on the alphabetical ordering of their names (i.e. `pre_plasticity` will be executed before `pre_transmission`). To explicitly specify the order, set the `order` attribute of the pathway, e.g.:

```
S.pre_transmission.order = -2
```

will make sure that the `pre_transmission` code is executed before the `pre_plasticity` code in each time step.

Multiple pathways can also be useful for abstract models of synaptic currents, e.g. modelling them as rectangular currents:

```
synapses = Synapses(...,
                    on_pre={'up': 'I_syn_post += 1*nA',
                            'down': 'I_syn_post -= 1*nA'},
                    delays={'up': 0*ms, 'down': 5*ms}  # 5ms-wide rectangular current
                    )
```

3.7.11 Numerical integration

Differential equation flags

For the integration of differential equations, one can use the same keywords as for *NeuronGroup*.

Note: Declaring a subexpression as `(constant over dt)` means that it will be evaluated each timestep for all synapses, potentially a very costly operation.

Explicit event-driven updates

As mentioned above, it is possible to write event-driven update code for the synaptic variables. This can also be done manually, by defining the variable `lastupdate` and referring to the predefined variable `t` (current time). Here's an example for short-term plasticity – but note that using the automatic `event-driven` approach from above is usually preferable:

```
S=Synapses(neuron_input,neuron,
           model='''x : 1
                    u : 1
                    w : 1
                    lastupdate : second''',
           on_pre='''u=U+(u-U)*exp(-(t-lastupdate)/tauf)
                     x=1+(x-1)*exp(-(t-lastupdate)/taud)
                     i+=w*u*x
                     x*=(1-u)
                     u+=U*(1-u)
                     lastupdate = t''')
```

By default, the `pre` pathway is executed before the `post` pathway (both are executed in the `'synapses'` scheduling slot, but the `pre` pathway has the `order` attribute -1, wheras the `post` pathway has `order` 1. See *Scheduling* for more details).

3.7.12 Technical notes

How connection arguments are interpreted

If conditions for connecting neurons are combined with both the `n` (number of synapses to create) and the `p` (probability of a synapse) keywords, they are interpreted in the following way:

> For every pair i, j:
>> if condition(i, j) is fulfilled:
>>> Evaluate p(i, j)
>>> If uniform random number between 0 and 1 < p(i, j):
>>>> Create n(i, j) synapses for (i, j)

With the generator syntax `j='EXPR for VAR in RANGE if COND'` (where the `RANGE` can be a full range or a random sample as described above), the interpretation is:

> For every i:
>> for every VAR in RANGE:
>>> j = EXPR
>>> if COND:
>>>> Create n(i, j) synapses for (i, j)

Note that the arguments in `RANGE` can only depend on `i` and the values of presynaptic variables. Similarly, the expression for `j`, `EXPR` can depend on `i`, presynaptic variables, and on the iteration variable `VAR`. The condition `COND` can depend on anything (presynaptic and postsynaptic variables).

The generator syntax expressing `i` as a function of `j` is interpreted in the same way:

> For every j:
>> for every VAR in RANGE:
>>> i = EXPR
>>> if COND:
>>>> Create n(i, j) synapses for (i, j)

Here, `RANGE` can only depend on `j` and postsynaptic variables, and `EXPR` can only depend on `j`, postsynaptic variables, and on the iteration variable `VAR`.

With the 1-to-1 mapping syntax `j='EXPR'` the interpretation is:

> For every i:
>> j = EXPR
>> Create n(i, j) synapses for (i, j)

And finally, `i='EXPR'` is interpreted as:

> For every j:
>> i = EXPR
>> Create n(i, j) synapses for (i, j)

Efficiency considerations

If you are connecting a single pair of neurons, the direct form `connect(i=5, j=10)` is the most efficient. However, if you are connecting a number of neurons, it will usually be more efficient to construct an array of `i` and `j` values and have a single `connect(i=i, j=j)` call.

For large connections, you should use one of the string based syntaxes where possible as this will generate compiled low-level code that will be typically much faster than equivalent Python code.

If you are expecting a majority of pairs of neurons to be connected, then using the condition-based syntax is optimal, e.g. `connect(condition='i!=j')`. However, if relatively few neurons are being connected then the 1-to-1 mapping or generator syntax will be better. For 1-to-1, `connect(j='i')` will always be faster than `connect(condition='i==j')` because the latter has to evaluate all `N**2` pairs `(i, j)` and check if the condition is true, whereas the former only has to do O(N) operations.

One tricky problem is how to efficiently generate connectivity with a probability `p(i, j)` that depends on both i and j, since this requires `N*N` computations even if the expected number of synapses is proportional to N. Some tricks for getting around this are shown in *Example: efficient_gaussian_connectivity*.

3.8 Input stimuli

For Brian 1 users

See the document *Inputs (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

- *Poisson inputs*
- *Spike generation*
- *Explicit equations*
- *Timed arrays*
- *Regular operations*
- *More on Poisson inputs*
- *Arbitrary Python code (network operations)*

There are various ways of providing "external" input to a network.

3.8.1 Poisson inputs

For generating spikes according to a Poisson point process, `PoissonGroup` can be used, e.g.:

```
P = PoissonGroup(100, np.arange(100)*Hz + 10*Hz)
G = NeuronGroup(100, 'dv/dt = -v / (10*ms) : 1')
S = Synapses(P, G, on_pre='v+=0.1')
S.connect(j='i')
```

See *More on Poisson inputs* below for further information.

For simulations where the individually generated spikes are just used as a source of input to a neuron, the `PoissonInput` class provides a more efficient alternative: see *Efficient Poisson inputs via PoissonInput* below for details.

3.8.2 Spike generation

You can also generate an explicit list of spikes given via arrays using *SpikeGeneratorGroup*. This object behaves just like a *NeuronGroup* in that you can connect it to other groups via a *Synapses* object, but you specify three bits of information: N the number of neurons in the group; indices an array of the indices of the neurons that will fire; and times an array of the same length as indices with the times that the neurons will fire a spike. The indices and times arrays are matching, so for example indices=[0,2,1] and times=[1*ms,2*ms,3*ms] means that neuron 0 fires at time 1 ms, neuron 2 fires at 2 ms and neuron 1 fires at 3 ms. Example use:

```
indices = array([0, 2, 1])
times = array([1, 2, 3])*ms
G = SpikeGeneratorGroup(3, indices, times)
```

The spikes that will be generated by *SpikeGeneratorGroup* can be changed between runs with the *set_spikes* method. This can be useful if the input to a system should depend on its previous output or when running multiple trials with different input:

```
inp = SpikeGeneratorGroup(N, indices, times)
G = NeuronGroup(N, '...')
feedforward = Synapses(inp, G, '...', on_pre='...')
feedforward.connect(j='i')
recurrent = Synapses(G, G, '...', on_pre='...')
recurrent.connect('i!=j')
spike_mon = SpikeMonitor(G)
# ...
run(runtime)
# Replay the previous output of group G as input into the group
inp.set_spikes(spike_mon.i, spike_mon.t + runtime)
run(runtime)
```

3.8.3 Explicit equations

If the input can be explicitly expressed as a function of time (e.g. a sinusoidal input current), then its description can be directly included in the equations of the respective group:

```
G = NeuronGroup(100, '''dv/dt = (-v + I)/(10*ms) : 1
                        rates : Hz  # each neuron's input has a different rate
                        size : 1  # and a different amplitude
                        I = size*sin(2*pi*rates*t) : 1''')
G.rates = '10*Hz + i*Hz'
G.size = '(100-i)/100. + 0.1'
```

3.8.4 Timed arrays

If the time dependence of the input cannot be expressed in the equations in the way shown above, it is possible to create a *TimedArray*. This acts as a function of time where the values at given time points are given explicitly. This can be especially useful to describe non-continuous stimulation. For example, the following code defines a *TimedArray* where stimulus blocks consist of a constant current of random strength for 30ms, followed by no stimulus for 20ms. Note that in this particular example, numerical integration can use exact methods, since it can assume that the *TimedArray* is a constant function of time during a single integration time step.

Note: The semantics of *TimedArray* changed slightly compared to Brian 1: for TimedArray([x1, x2, ...], dt=my_dt), the value x1 will be returned for all 0<=t<my_dt, x2 for my_dt<=t<2*my_dt etc., whereas Brian1

---

returned `x1` for `0<=t<0.5*my_dt`, `x2` for `0.5*my_dt<=t<1.5*my_dt`, etc.

```
stimulus = TimedArray(np.hstack([[c, c, c, 0, 0]
                                 for c in np.random.rand(1000)]),
                      dt=10*ms)
G = NeuronGroup(100, 'dv/dt = (-v + stimulus(t))/(10*ms) : 1',
                threshold='v>1', reset='v=0')
G.v = '0.5*rand()'  # different initial values for the neurons
```

TimedArray can take a one-dimensional value array (as above) and therefore return the same value for all neurons or it can take a two-dimensional array with time as the first and (neuron/synapse/…-)index as the second dimension.

In the following, this is used to implement shared noise between neurons, all the "even neurons" get the first noise instantiation, all the "odd neurons" get the second:

```
runtime = 1*second
stimulus = TimedArray(np.random.rand(int(runtime/defaultclock.dt), 2),
                      dt=defaultclock.dt)
G = NeuronGroup(100, 'dv/dt = (-v + stimulus(t, i % 2))/(10*ms) : 1',
                threshold='v>1', reset='v=0')
```

3.8.5 Regular operations

An alternative to specifying a stimulus in advance is to run explicitly specified code at certain points during a simulation. This can be achieved with *run_regularly()*. One can think of these statements as equivalent to reset statements but executed unconditionally (i.e. for all neurons) and possibly on a different clock than the rest of the group. The following code changes the stimulus strength of half of the neurons (randomly chosen) to a new random value every 50ms. Note that the statement uses logical expressions to have the values only updated for the chosen subset of neurons (where the newly introduced auxiliary variable `change` equals 1):

```
G = NeuronGroup(100, '''dv/dt = (-v + I)/(10*ms) : 1
                        I : 1  # one stimulus per neuron''')
G.run_regularly('''change = int(rand() < 0.5)
                   I = change*(rand()*2) + (1-change)*I''',
                dt=50*ms)
```

The following topics are not essential for beginners.

3.8.6 More on Poisson inputs

Setting rates for Poisson inputs

`PoissonGroup` takes either a constant rate, an array of rates (one rate per neuron, as in the example above), or a string expression evaluating to a rate as an argument.

If the given value for `rates` is a constant, then using `PoissonGroup(N, rates)` is equivalent to:

```
NeuronGroup(N, 'rates : Hz', threshold='rand()<rates*dt')
```

and setting the group's `rates` attribute.

If `rates` is a string, then this is equivalent to:

```
NeuronGroup(N, 'rates = ... : Hz', threshold='rand()<rates*dt')
```

with the respective expression for the rates. This expression will be evaluated at every time step and therefore allows the use of time-dependent rates, i.e. inhomogeneous Poisson processes. For example, the following code (see also *Timed arrays*) uses a *TimedArray* to define the rates of a *PoissonGroup* as a function of time, resulting in five 100ms blocks of 100 Hz stimulation, followed by 100ms of silence:

```
stimulus = TimedArray(np.tile([100., 0.], 5)*Hz, dt=100.*ms)
P = PoissonGroup(1, rates='stimulus(t)')
```

Note that, as can be seen in its equivalent *NeuronGroup* formulation, a *PoissonGroup* does not work for high rates where more than one spike might fall into a single timestep. Use several units with lower rates in this case (e.g. use `PoissonGroup(10, 1000*Hz)` instead of `PoissonGroup(1, 10000*Hz)`).

Efficient Poisson inputs via PoissonInput

For simulations where the *PoissonGroup* is just used as a source of input to a neuron (i.e., the individually generated spikes are not important, just their impact on the target cell), the *PoissonInput* class provides a more efficient alternative: instead of generating spikes, *PoissonInput* directly updates a target variable based on the sum of independent Poisson processes:

```
G = NeuronGroup(100, 'dv/dt = -v / (10*ms) : 1')
P = PoissonInput(G, 'v', 100, 100*Hz, weight=0.1)
```

Each input of the *PoissonInput* is connected to all the neurons of the target *NeuronGroup* but each neuron receives independent realizations of the Poisson spike trains. Note that the *PoissonInput* class is however more restrictive than *PoissonGroup*, it only allows for a constant rate across all neurons (but you can create several *PoissonInput* objects, targeting different subgroups). It internally uses *BinomialFunction* which will draw a random number each time step, either from a binomial distribution or from a normal distribution as an approximation to the binomial distribution if $np > 5 \wedge n(1-p) > 5$, where n is the number of inputs and $p = dt \cdot rate$ the spiking probability for a single input.

3.8.7 Arbitrary Python code (network operations)

If none of the above techniques is general enough to fulfill the requirements of a simulation, Brian allows you to write a *NetworkOperation*, an arbitrary Python function that is executed every time step (possible on a different clock than the rest of the simulation). This function can do arbitrary operations, use conditional statements etc. and it will be executed as it is (i.e. as pure Python code even if cython code generation is active). Note that one cannot use network operations in combination with the C++ standalone mode. Network operations are particularly useful when some condition or calculation depends on operations across neurons, which is currently not possible to express in abstract code. The following code switches input on for a randomly chosen single neuron every 50 ms:

```python
G = NeuronGroup(10, '''dv/dt = (-v + active*I)/(10*ms) : 1
                       I = sin(2*pi*100*Hz*t) : 1 (shared) #single input
                       active : 1  # will be set in the network operation''')
@network_operation(dt=50*ms)
def update_active():
    index = np.random.randint(10)  # index for the active neuron
    G.active_ = 0  # the underscore switches off unit checking
    G.active_[index] = 1
```

Note that the network operation (in the above example: `update_active`) has to be included in the *Network* object if one is constructed explicitly.

Only functions with zero or one arguments can be used as a *NetworkOperation*. If the function has one argument then it will be passed the current time `t`:

```python
@network_operation(dt=1*ms)
def update_input(t):
    if t>50*ms and t<100*ms:
        pass # do something
```

Note that this is preferable to accessing `defaultclock.t` from within the function – if the network operation is not running on the *defaultclock* itself, then that value is not guaranteed to be correct.

Instance methods can be used as network operations as well, however in this case they have to be constructed explicitly, the *network_operation()* decorator cannot be used:

```python
class Simulation(object):
    def __init__(self, data):
        self.data = data
        self.group = NeuronGroup(...)
        self.network_op = NetworkOperation(self.update_func, dt=10*ms)
        self.network = Network(self.group, self.network_op)

    def update_func(self):
        pass # do something

    def run(self, runtime):
        self.network.run(runtime)
```

3.9 Recording during a simulation

> **For Brian 1 users**
>
> See the document *Monitors (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

> - *Recording spikes*
> - *Recording variables at spike time*
> - *Recording variables continuously*
> - *Recording population rates*
> - *Getting all data*
> - *Recording values for a subset of the run*
> - *Freeing up memory in long recordings*
> - *Recording random subsets of neurons*

Recording variables during a simulation is done with "monitor" objects. Specifically, spikes are recorded with *Spike-Monitor*, the time evolution of variables with *StateMonitor* and the firing rate of a population of neurons with *PopulationRateMonitor*.

3.9.1 Recording spikes

To record spikes from a group `G` simply create a *SpikeMonitor* via `SpikeMonitor(G)`. After the simulation, you can access the attributes `i`, `t`, `num_spikes` and `count` of the monitor. The `i` and `t` attributes give the array of neuron indices and times of the spikes. For example, if `M.i==[0, 2, 1]` and `M.t==[1*ms, 2*ms, 3*ms]` it means that neuron 0 fired a spike at 1 ms, neuron 2 fired a spike at 2 ms, and neuron 1 fired a spike at 3 ms. Alternatively, you can also call the *spike_trains* method to get a dictionary mapping neuron indices to arrays of spike times, i.e. in the above example, `spike_trains = M.spike_trains(); spike_trains[1]` would return `array([3.]) * msecond`. The `num_spikes` attribute gives the total number of spikes recorded, and `count` is an array of the length of the recorded group giving the total number of spikes recorded from each neuron.

Example:

```
G = NeuronGroup(N, model='...')
M = SpikeMonitor(G)
run(runtime)
plot(M.t/ms, M.i, '.')
```

If you are only interested in summary statistics but not the individual spikes, you can set the `record` argument to `False`. You will then not have access to `i` and `t` but you can still get the `count` and the total number of spikes (`num_spikes`).

3.9.2 Recording variables at spike time

By default, a *SpikeMonitor* only records the time of the spike and the index of the neuron that spiked. Sometimes it can be useful to addtionaly record other variables, e.g. the membrane potential for models where the threshold is not at a fixed value. This can be done by providing an extra `variables` argument, the recorded variable can then be accessed as an attribute of the *SpikeMonitor*, e.g.:

```
G = NeuronGroup(10, 'v : 1', threshold='rand()<100*Hz*dt')
G.run_regularly('v = rand()')
M = SpikeMonitor(G, variables=['v'])
run(100*ms)
plot(M.t/ms, M.v, '.')
```

To conveniently access the values of a recorded variable for a single neuron, the *SpikeMonitor.values* method can be used that returns a dictionary with the values for each neuron.:

```
G = NeuronGroup(N, '''dv/dt = (1-v)/(10*ms) : 1
                      v_th : 1''',
                threshold='v > v_th',
                # randomly change the threshold after a spike:
                reset='''v=0
                         v_th = clip(v_th + rand()*0.2 - 0.1, 0.1, 0.9)''')
G.v_th = 0.5
spike_mon = SpikeMonitor(G, variables='v')
run(1*second)
v_values = spike_mon.values('v')
print('Threshold crossing values for neuron 0: {}'.format(v_values[0]))
hist(spike_mon.v, np.arange(0, 1, .1))
show()
```

> **Note:** Spikes are not the only events that can trigger recordings, see *Custom events*.

3.9.3 Recording variables continuously

To record how a variable evolves over time, use a *StateMonitor*, e.g. to record the variable v at every time step and plot it for neuron 0:

```
G = NeuronGroup(...)
M = StateMonitor(G, 'v', record=True)
run(...)
plot(M.t/ms, M.v[0]/mV)
```

In general, you specify the group, variables and indices you want to record from. You specify the variables with a string or list of strings, and the indices either as an array of indices or `True` to record all indices (but beware because this may take a lot of memory).

After the simulation, you can access these variables as attributes of the monitor. They are 2D arrays with shape `(num_indices, num_times)`. The special attribute `t` is an array of length `num_times` with the corresponding times at which the values were recorded.

Note that you can also use *StateMonitor* to record from *Synapses* where the indices are the synapse indices rather than neuron indices.

In this example, we record two variables v and u, and record from indices 0, 10 and 100. Afterwards, we plot the recorded values of v and u from neuron 0:

```
G = NeuronGroup(...)
M = StateMonitor(G, ('v', 'u'), record=[0, 10, 100])
run(...)
plot(M.t/ms, M.v[0]/mV, label='v')
plot(M.t/ms, M.u[0]/mV, label='u')
```

There are two subtly different ways to get the values for specific neurons: you can either index the 2D array stored in the attribute with the variable name (as in the example above) or you can index the monitor itself. The former will use an index relative to the recorded neurons (e.g. `M.v[1]` will return the values for the second *recorded* neuron which is the neuron with the index 10 whereas `M.v[10]` would raise an error because only three neurons have been recorded), whereas the latter will use an absolute index corresponding to the recorded group (e.g. `M[1].v` will raise an error because the neuron with the index 1 has not been recorded and `M[10].v` will return the values for the neuron with the index 10). If all neurons have been recorded (e.g. with `record=True`) then both forms give the same result.

Note that for plotting all recorded values at once, you have to transpose the variable values:

```
plot(M.t/ms, M.v.T/mV)
```

Note: In contrast to Brian 1, the values are recorded at the beginning of a time step and not at the end (you can set the `when` argument when creating a *StateMonitor*, details about scheduling can be found here: *Custom progress reporting*).

3.9.4 Recording population rates

To record the time-varying firing rate of a population of neurons use *PopulationRateMonitor*. After the simulation the monitor will have two attributes `t` and `rate`, the latter giving the firing rate at each time step corresponding to the time in `t`. For example:

```
G = NeuronGroup(...)
M = PopulationRateMonitor(G)
run(...)
plot(M.t/ms, M.rate/Hz)
```

To get a smoother version of the rate, use *PopulationRateMonitor.smooth_rate*.

The following topics are not essential for beginners.

3.9.5 Getting all data

Note that all monitors are implement as "groups", so you can get all the stored values in a monitor with the `get_states` method, which can be useful to dump all recorded data to disk, for example:

```python
import pickle
group = NeuronGroup(...)
state_mon = StateMonitor(group, 'v', record=...)
run(...)
data = state_mon.get_states(['t', 'v'])
with open('state_mon.pickle', 'w') as f:
    pickle.dump(data, f)
```

3.9.6 Recording values for a subset of the run

Monitors can be created and deleted between runs, e.g. to ignore the first second of your simulation in your recordings you can do:

```python
# Set up network without monitor
run(1*second)
state_mon = StateMonitor(....)
run(...)  # Continue run and record with the StateMonitor
```

Alternatively, you can set the monitor's `active` attribute as explained in the *Scheduling* section.

3.9.7 Freeing up memory in long recordings

Creating and deleting monitors can also be useful to free memory during a long recording. The following will do a simulation run, dump the monitor data to disk, delete the monitor and finally continue the run with a new monitor:

```python
import pickle
# Set up network
state_mon = StateMonitor(...)
run(...)  # a long run
data = state_mon.get_states(...)
with open('first_part.data', 'w') as f:
    pickle.dump(data, f)
del state_mon
del data
state_mon = StateMonitor(...)
run(...)  # another long run
```

Note that this technique cannot be applied in *standalone mode*.

3.9.8 Recording random subsets of neurons

In large networks, you might only be interested in the activity of a random subset of neurons. While you can specify a `record` argument for a *StateMonitor* that allows you to select a subset of neurons, this is not possible for *SpikeMonitor*/*EventMonitor* and *PopulationRateMonitor*. However, Brian allows you to record with these monitors from a subset of neurons by using a *subgroup*:

```
group = NeuronGroup(1000, ...)
spike_mon = SpikeMonitor(group[:100])  # only record first 100 neurons
```

It might seem like a restriction that such a subgroup has to be contiguous, but the order of neurons in a group does not have any meaning as such; in a randomly ordered group of neurons, any contiguous group of neurons can be considered a random subset. If some aspects of your model *do* depend on the position of the neuron in a group (e.g. a ring model, where neurons are connected based on their distance in the ring, or a model where initial values or parameters span a range of values in a regular fashion), then this requires an extra step: instead of using the order of neurons in the group directly, or depending on the neuron index `i`, create a new, shuffled, index variable as part of the model definition and then depend on this index instead:

```
group = NeuronGroup(10000, '''....
                                index : integer (constant)''')
indices = group.i[:]
np.random.shuffle(indices)
group.index = indices
# Then use 'index' in string expressions or use it as an index array
# for initial values/parameters defined as numpy arrays
```

If this solution is not feasible for some reason, there is another approach that works for a *SpikeMonitor*/*EventMonitor*. You can add an additional flag to each neuron, stating whether it should be recorded or not. Then, you define a new *custom event* that is identical to the event you are interested in, but additionally requires the flag to be set. E.g. to only record the spikes of neurons with the `to_record` attribute set:

```
group = NeuronGroup(..., '''...
                            to_record : boolean (constant)''',
                    threshold='...', reset='...',
                    events={'recorded_spike': '... and to_record'})
group.to_record = ...
mon_events = EventMonitor(group, 'recorded_spike')
```

Note that this solution will evaluate the threshold condition for each neuron twice, and is therefore slightly less efficient. There's one additional caveat: you'll have to manually include `and not_refractory` in your `events` definition if your neuron uses refractoriness. This is done automatically for the `threshold` condition, but not for any user-defined events.

3.10 Running a simulation

For Brian 1 users

See the document *Networks and clocks (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

- *Networks*

To run a simulation, one either constructs a new `Network` object and calls its `Network.run` method, or uses the "magic" system and a plain `run()` call, collecting all the objects in the current namespace.

Note that Brian has several different ways of running the actual computations, and choosing the right one can make orders of magnitude of difference in terms of simplicity and efficiency. See *Computational methods and efficiency* for more details.

3.10.1 Networks

In most straightforward simulations, you do not have to explicitly create a `Network` object but instead can simply call `run()` to run a simulation. This is what is called the "magic" system, because Brian figures out automatically what you want to do.

When calling `run()`, Brian runs the `collect()` function to gather all the objects in the current context. It will include all the objects that are "visible", i.e. that you could refer to with an explicit name:

```
G = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                threshold='v > 1', reset='v = 0')
S = Synapses(G, G, model='w:1', on_pre='v+=w')
S.connect('i!=j')
S.w = 'rand()'
mon = SpikeMonitor(G)

run(10*ms)   # will include G, S, mon
```

Note that it will not automatically include objects that are "hidden" in containers, e.g. if you store several monitors in a list. Use an explicit `Network` object in this case. It might be convenient to use the `collect()` function when creating the `Network` object in that case:

```
G = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                threshold='v > 1', reset='v = 0')
S = Synapses(G, G, model='w:1', on_pre='v+=w')
S.connect('i!=j')
S.w = 'rand()'
monitors = [SpikeMonitor(G), StateMonitor(G, 'v', record=True)]

# a simple run would not include the monitors
net = Network(collect())   # automatically include G and S
net.add(monitors)   # manually add the monitors

net.run(10*ms)
```

3.10.2 Setting the simulation time step

To set the simulation time step for every simulated object, set the `dt` attribute of the *defaultclock* which is used by all objects that do not explicitly specify a `clock` or `dt` value during construction:

```
defaultclock.dt = 0.05*ms
```

If some objects should use a different clock (e.g. to record values with a *StateMonitor* not at every time step in a long running simulation), you can provide a `dt` argument to the respective object:

```
s_mon = StateMonitor(group, 'v', record=True, dt=1*ms)
```

To sum up:

- Set `defaultclock.dt` to the time step that should be used by most (or all) of your objects.
- Set `dt` explicitly when creating objects that should use a different time step.

Behind the scenes, a new *Clock* object will be created for each object that defines its own `dt` value.

3.10.3 Progress reporting

Especially for long simulations it is useful to get some feedback about the progress of the simulation. Brian offers a few built-in options and an extensible system to report the progress of the simulation. In the *Network.run* or *run()* call, two arguments determine the output: `report` and `report_period`. When `report` is set to `'text'` or `'stdout'`, the progress will be printed to the standard output, when it is set to `'stderr'`, it will be printed to "standard error". There will be output at the start and the end of the run, and during the run in `report_period` intervals. It is also possible to do *custom progress reporting*.

3.10.4 Continuing/repeating simulations

To store the current state of the simulation, call *store()* (use the *Network.store* method for a *Network*). You can store more than one snapshot of a system by providing a name for the snapshot; if *store()* is called without a specified name, `'default'` is used as the name. To restore the state, use *restore()*.

The following simple example shows how this system can be used to run several trials of an experiment:

```
# set up the network
G = NeuronGroup(...)
...
spike_monitor = SpikeMonitor(G)

# Snapshot the state
store()

# Run the trials
spike_counts = []
for trial in range(3):
    restore()  # Restore the initial state
    run(...)
    # store the results
    spike_counts.append(spike_monitor.count)
```

The following schematic shows how multiple snapshots can be used to run a network with a separate "train" and "test" phase. After training, the test is run several times based on the trained network. The whole process of training and testing is repeated several times as well:

```python
# set up the network
G = NeuronGroup(..., '''...
                    test_input : amp
                    ...''')
S = Synapses(..., '''...
                    plastic : boolean (shared)
                    ...''')
G.v = ...
S.connect(...)
S.w = ...

# First snapshot at t=0
store('initialized')

# Run 3 complete trials
for trial in range(3):
    # Simulate training phase
    restore('initialized')
    S.plastic = True
    run(...)

    # Snapshot after learning
    store('after_learning')

    # Run 5 tests after the training
    for test_number in range(5):
        restore('after_learning')
        S.plastic = False  # switch plasticity off
        G.test_input = test_inputs[test_number]
        # monitor the activity now
        spike_mon = SpikeMonitor(G)
        run(...)
        # Do something with the result
        # ...
```

The following topics are not essential for beginners.

3.10.5 Multiple magic runs

When you use more than a single `run()` statement, the magic system tries to detect which of the following two situations applies:

1. You want to continue a previous simulation

2. You want to start a new simulation

For this, it uses the following heuristic: if a simulation consists only of objects that have not been run, it will start a new simulation starting at time 0 (corresponding to the creation of a new `Network` object). If a simulation only consists of objects that have been simulated in the previous `run()` call, it will continue that simulation at the previous time.

If neither of these two situations apply, i.e., the network consists of a mix of previously run objects and new objects, an error will be raised. If this is not a mistake but intended (e.g. when a new input source and synapses should be added to a network at a later stage), use an explicit `Network` object.

In these checks, "non-invalidating" objects (i.e. objects that have `BrianObject.invalidates_magic_network` set to `False`) are ignored, e.g. creating new monitors is always possible.

Note that if you do not want to run an object for the complete duration of your simulation, you can create the object in the beginning of your simulation and then set its `active` attribute. For details, see the *Scheduling* section below.

3.10.6 Changing the simulation time step

You can change the simulation time step after objects have been created or even after a simulation has been run:

```
defaultclock.dt = 0.1*ms
# Set the network
# ...
run(initial_time)
defaultclock.dt = 0.01*ms
run(full_time - initial_time)
```

To change the time step between runs for objects that do not use the `defaultclock`, you cannot directly change their `dt` attribute (which is read-only) but instead you have to change the `dt` of the `clock` attribute. If you want to change the `dt` value of several objects at the same time (but not for all of them, i.e. when you cannot use `defaultclock.dt`) then you might consider creating a `Clock` object explicitly and then passing this clock to each object with the `clock` keyword argument (instead of `dt`). This way, you can later change the `dt` for several objects at once by assigning a new value to `Clock.dt`.

Note that a change of `dt` has to be compatible with the internal representation of clocks as an integer value (the number of elapsed time steps). For example, you can simulate an object for 100ms with a time step of 0.1ms (i.e. for 1000 steps) and then switch to a `dt` of 0.5ms, the time will then be internally represented as 200 steps. You cannot, however, switch to a dt of 0.3ms, because 100ms are not an integer multiple of 0.3ms.

3.10.7 Profiling

To get an idea which parts of a simulation take the most time, Brian offers a basic profiling mechanism. If a simulation is run with the `profile=True` keyword argument, it will collect information about the total simulation time for each `CodeObject`. This information can then be retrieved from `Network.profiling_info`, which contains a list of `(name, time)` tuples or a string summary can be obtained by calling `profiling_summary()`. The following example shows profiling output after running the CUBA example (where the neuronal state updates take up the most time):

```
>>> profiling_summary(show=5)  # show the 5 objects that took the longest
Profiling summary
=================
neurongroup_stateupdater    5.54 s    61.32 %
synapses_pre                1.39 s    15.39 %
synapses_1_pre              1.03 s    11.37 %
spikemonitor                0.59 s     6.55 %
neurongroup_thresholder     0.33 s     3.66 %
```

3.10.8 Scheduling

Every simulated object in Brian has three attributes that can be specified at object creation time: `dt`, `when`, and `order`. The time step of the simulation is determined by `dt`, if it is specified, or otherwise by `defaultclock.dt`. Changing this will therefore change the `dt` of all objects that don't specify one. Alternatively, a `clock` object can be specified directly, this can be useful if a clock should be shared between several objects – under most circumstances, however, a user should not have to deal with the creation of *Clock* objects and just define `dt`.

During a single time step, objects are updated in an order according first to their `when` argument's position in the schedule. This schedule is determined by *Network.schedule* which is a list of strings, determining "execution slots" and their order. It defaults to: `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`. In addition to the names provided in the schedule, names such as `before_thresholds` or `after_synapses` can be used that are understood as slots in the respective positions. The default for the `when` attribute is a sensible value for most objects (resets will happen in the `reset` slot, etc.) but sometimes it make sense to change it, e.g. if one would like a *StateMonitor*, which by default records in the `start` slot, to record the membrane potential before a reset is applied (otherwise no threshold crossings will be observed in the membrane potential traces).

Finally, if during a time step two objects fall in the same execution slot, they will be updated in ascending order according to their `order` attribute, an integer number defaulting to 0. If two objects have the same `when` and `order` attribute then they will be updated in an arbitrary but reproducible order (based on the lexicographical order of their names).

Note that objects that don't do any computation by themselves but only act as a container for other objects (e.g. a *NeuronGroup* which contains a `StateUpdater`, a `Resetter` and a `Thresholder`), don't have any value for `when`, but pass on the given values for `dt` and `order` to their containing objects.

If you want your simulation object to run only for a particular time period of the whole simulation, you can use the *active* attribute. For example, this can be useful when you want a monitor to be active only for some time out of a long simulation:

```
# Set up the network
# ...
monitor = SpikeMonitor(...)
monitor.active = False
run(long_time*seconds)  # not recording
monitor.active = True
run(required_time*seconds)  # recording
```

To see how the objects in a network are scheduled, you can use the *scheduling_summary()* function:

```
>>> group = NeuronGroup(10, 'dv/dt = -v/(10*ms) : 1', threshold='v > 1',
...                     reset='v = 0')
>>> mon = StateMonitor(group, 'v', record=True, dt=1*ms)
>>> scheduling_summary()
             object                   |            part of            |       Clock␣
↪dt        |    when    | order | active
--------------------------------------+-------------------------------+--------------
↪---------+-----------+-------+-------
statemonitor (StateMonitor)           | statemonitor (StateMonitor) | 1. ms (every␣
↪10 steps) | start      |     0 | yes
neurongroup_stateupdater (StateUpdater) | neurongroup (NeuronGroup)   | 100. us␣
↪(every step)    | groups     |     0 | yes
neurongroup_thresholder (Thresholder) | neurongroup (NeuronGroup)   | 100. us␣
↪(every step)    | thresholds |     0 | yes
neurongroup_resetter (Resetter)       | neurongroup (NeuronGroup)   | 100. us␣
↪(every step)    | resets     |     0 | yes
```

As you can see in the output above, the *StateMonitor* will only record the membrane potential every 10 time steps, but when it does, it will do it at the start of the time step, before the numerical integration, the thresholding, and the reset

---

operation takes place.

Every new `Network` starts a simulation at time 0; `Network.t` is a read-only attribute, to go back to a previous moment in time (e.g. to do another trial of a simulation with a new noise instantiation) use the mechanism described below.

3.10.9 Store/restore

Note that `Network.run`, `Network.store` and `Network.restore` (or `run()`, `store()`, `restore()`) are the only way of affecting the time of the clocks. In contrast to Brian1, it is no longer necessary (nor possible) to directly set the time of the clocks or call a `reinit` function.

The state of a network can also be stored on disk with the optional `filename` argument of `Network.store`/`store()`. This way, you can run the initial part of a simulation once, store it to disk, and then continue from this state later. Note that the `store()`/`restore()` mechanism does not re-create the network as such, you still need to construct all the `NeuronGroup`, `Synapses`, `StateMonitor`, … objects, restoring will only restore all the state variable values (membrane potential, conductances, synaptic connections/weights/delays, …). This restoration does however restore the internal state of the objects as well, e.g. spikes that have not been delivered yet because of synaptic delays will be delivered correctly.

3.11 Multicompartment models

> **For Brian 1 users**
>
> See the document *Multicompartmental models (Brian 1 –> 2 conversion)* for details how to convert Brian 1 code.

It is possible to create neuron models with a spatially extended morphology, using the `SpatialNeuron` class. A `SpatialNeuron` is a single neuron with many compartments. Essentially, it works as a `NeuronGroup` where elements are compartments instead of neurons.

A `SpatialNeuron` is specified by a morphology (see *Creating a neuron morphology*) and a set of equations for trans-membrane currents (see *Creating a spatially extended neuron*).

3.11.1 Creating a neuron morphology

Schematic morphologies

Morphologies can be created combining geometrical objects:

```
soma = Soma(diameter=30*um)
cylinder = Cylinder(diameter=1*um, length=100*um, n=10)
```

The first statement creates a single iso-potential compartment (i.e. with no axial resistance within the compartment), with its area calculated as the area of a sphere with the given diameter. The second one specifies a cylinder consisting of 10 compartments with identical diameter and the given total length.

For more precise control over the geometry, you can specify the length and diameter of each individual compartment, including the diameter at the start of the section (i.e. for `n` compartments: `n` length and `n+1` diameter values) in a `Section` object:

```
section = Section(diameter=[6, 5, 4, 3, 2, 1]*um, length=[10, 10, 10, 5, 5]*um, n=5)
```

The individual compartments are modeled as truncated cones, changing the diameter linearly between the given diameters over the length of the compartment. Note that the `diameter` argument specifies the values at the nodes *between* the compartments, but accessing the `diameter` attribute of a `Morphology` object will return the diameter at the *center* of the compartment (see the note below).

The following table summarizes the different options to create schematic morphologies (the black compartment before the start of the section represents the parent compartment with diameter 15 μm, not specified in the code below):

	Example
Soma	```# Soma always has a single␣↪compartment``` ```Soma(diameter=30*um)```
Cylinder	```# Each compartment has fixed␣↪length and diameter``` ```Cylinder(n=5, diameter=10*um,␣↪length=50*um)```

Section	```# Length and diameter␣↪individually defined for␣↪each compartment (at start```
	 ```# and end)```
	 ```Section(n=5, diameter=[15, 5,␣↪10, 5, 10, 5]*um,```
	 ``` length=[10, 20, 5, 5,␣↪10]*um)```

Note: For a `Section`, the `diameter` argument specifies the diameter *between* the compartments (and at the beginning/end of the first/last compartment). the corresponding values can therefore be later retrieved from the `Morphology` via the `start_diameter` and `end_diameter` attributes. The `diameter` attribute of a `Morphology` does correspond to the diameter at the midpoint of the compartment. For a `Cylinder`, `start_diameter`, `diameter`, and `end_diameter` are of course all identical.

The tree structure of a morphology is created by attaching `Morphology` objects together:

```
morpho = Soma(diameter=30*um)
morpho.axon = Cylinder(length=100*um, diameter=1*um, n=10)
morpho.dendrite = Cylinder(length=50*um, diameter=2*um, n=5)
```

These statements create a morphology consisting of a cylindrical axon and a dendrite attached to a spherical soma. Note that the names `axon` and `dendrite` are arbitrary and chosen by the user. For example, the same morphology can be created as follows:

```
morpho = Soma(diameter=30*um)
morpho.output_process = Cylinder(length=100*um, diameter=1*um, n=10)
morpho.input_process = Cylinder(length=50*um, diameter=2*um, n=5)
```

The syntax is recursive, for example two sections can be added at the end of the dendrite as follows:

```
morpho.dendrite.branch1 = Cylinder(length=50*um, diameter=1*um, n=3)
morpho.dendrite.branch2 = Cylinder(length=50*um, diameter=1*um, n=3)
```

Equivalently, one can use an indexing syntax:

```
morpho['dendrite']['branch1'] = Cylinder(length=50*um, diameter=1*um, n=3)
morpho['dendrite']['branch2'] = Cylinder(length=50*um, diameter=1*um, n=3)
```

The names given to sections are completely up to the user. However, names that consist of a single digit (1 to 9) or the letters L (for left) and R (for right) allow for a special short syntax: they can be joined together directly, without the needs for dots (or dictionary syntax) and therefore allow to quickly navigate through the morphology tree (e.g. `morpho.LRLLR` is equivalent to `morpho.L.R.L.L.R`). This short syntax can also be used to create trees:

```
>>> morpho = Soma(diameter=30*um)
>>> morpho.L = Cylinder(length=10*um, diameter=1*um, n=3)
>>> morpho.L1 = Cylinder(length=5*um, diameter=1*um, n=3)
>>> morpho.L2 = Cylinder(length=5*um, diameter=1*um, n=3)
>>> morpho.L3 = Cylinder(length=5*um, diameter=1*um, n=3)
>>> morpho.R = Cylinder(length=10*um, diameter=1*um, n=3)
>>> morpho.RL = Cylinder(length=5*um, diameter=1*um, n=3)
>>> morpho.RR = Cylinder(length=5*um, diameter=1*um, n=3)
```

The above instructions create a dendritic tree with two main sections, three sections attached to the first section and two to the second. This can be verified with the *Morphology.topology* method:

```
>>> morpho.topology()
( )  [root]
   `---|   .L
         `---|   .L.1
         `---|   .L.2
         `---|   .L.3
   `---|   .R
         `---|   .R.L
         `---|   .R.R
```

Note that an expression such as `morpho.L` will always refer to the entire subtree. However, accessing the attributes (e.g. `diameter`) will only return the values for the given section.

Note: To avoid ambiguities, do not use names for sections that can be interpreted in the abbreviated way detailed above. For example, do not name a child section `L1` (which will be interpreted as the first child of the child `L`)

The number of compartments in a section can be accessed with `morpho.n` (or `morpho.L.n`, etc.), the number of total sections and compartments in a subtree can be accessed with `morpho.total_sections` and `morpho.total_compartments` respectively.

Adding coordinates

For plotting purposes, it can be useful to add coordinates to a *Morphology* that was created using the "schematic" approach described above. This can be done by calling the generate_coordinates method on a morphology, which will return an identical morphology but with additional 2D or 3D coordinates. By default, this method creates a morphology according to a deterministic algorithm in 2D:

```
new_morpho = morpho.generate_coordinates()
```
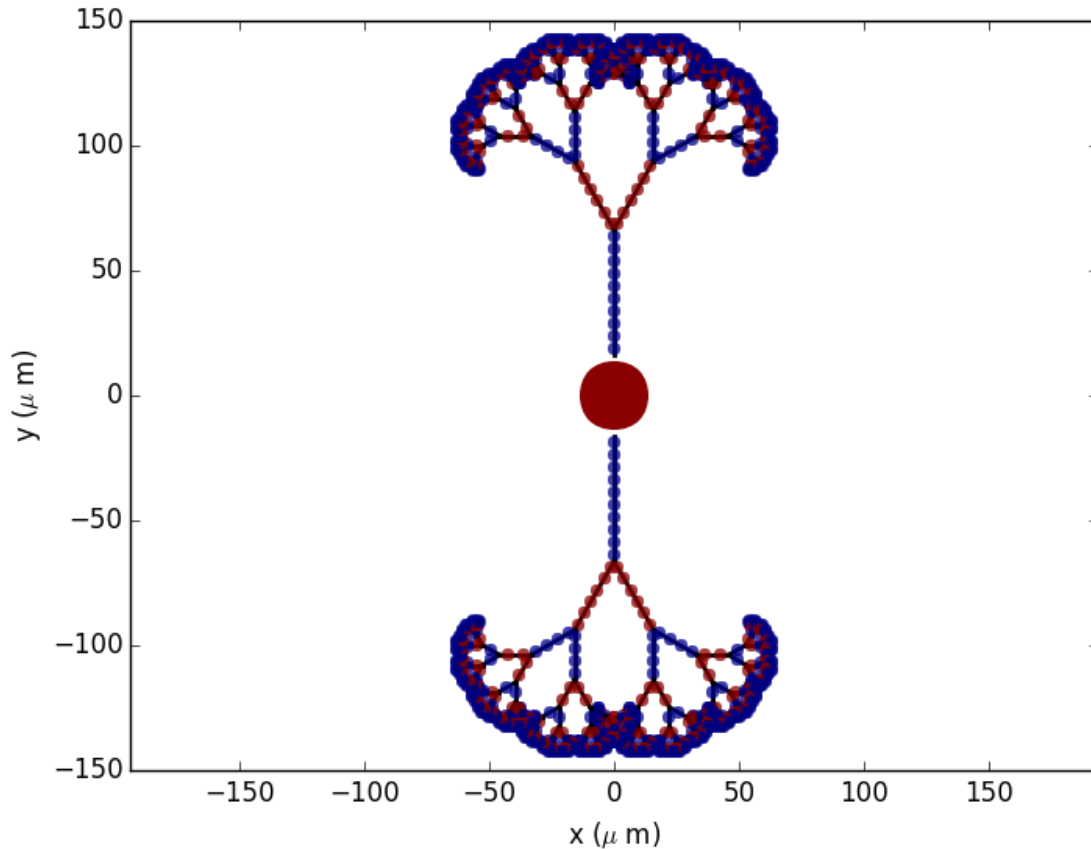

To get more "realistic" morphologies, this function can also be used to create morphologies in 3D where the orientation of each section differs from the orientation of the parent section by a random amount:

```
new_morpho = morpho.generate_coordinates(section_randomness=25)
```
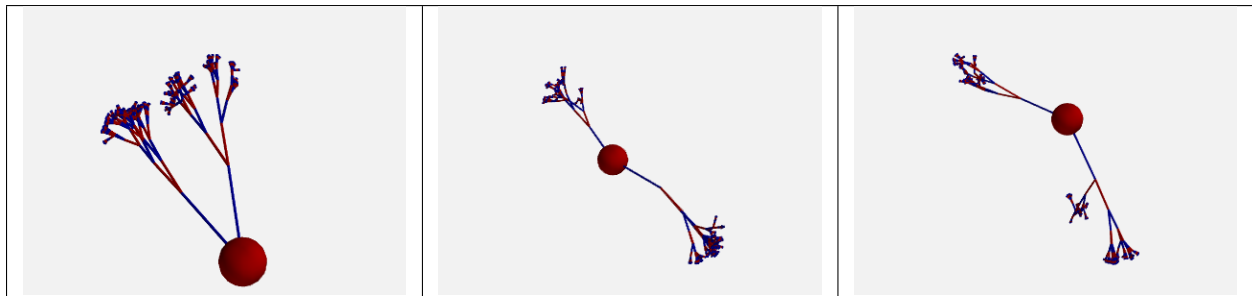
This algorithm will base the orientation of each section on the orientation of the parent section and then randomly perturb this orientation. More precisely, the algorithm first chooses a random vector orthogonal to the orientation of the parent section. Then, the section will be rotated around this orthogonal vector by a random angle, drawn from an exponential distribution with the β parameter (in degrees) given by `section_randomness`. This β parameter specifies both the mean and the standard deviation of the rotation angle. Note that no maximum rotation angle is enforced, values for `section_randomness` should therefore be reasonably small (e.g. using a `section_randomness` of `45` would already lead to a probability of ~14% that the section will be rotated by more than 90 degrees, therefore making the section go "backwards").

In addition, also the orientation of each compartment within a section can be randomly varied:

```
new_morpho = morpho.generate_coordinates(section_randomness=25,
                                         compartment_randomness=15)
```
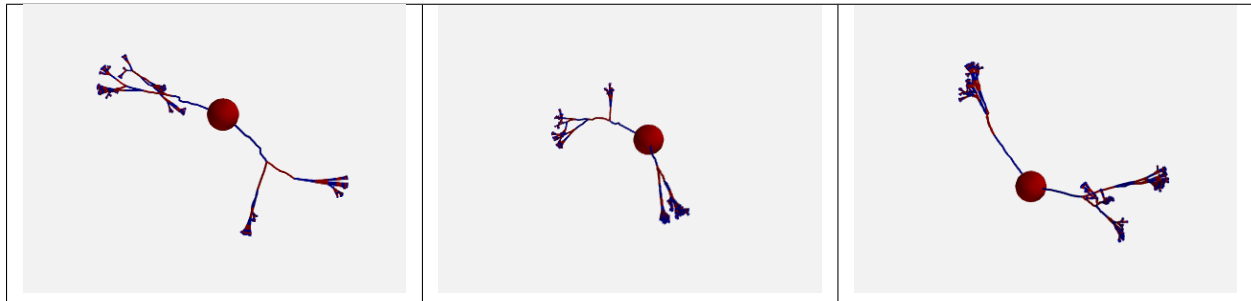

The algorithm is the same as the one presented above, but applied individually to each compartment within a section (still based on the orientation on the parent *section*, not on the orientation of the previous *compartment*).

Complex morphologies

Morphologies can also be created from information about the compartment coordinates in 3D space. Such morphologies can be loaded from a `.swc` file (a standard format for neuronal morphologies; for a large database of morphologies in this format see http://neuromorpho.org):

```
morpho = Morphology.from_file('corticalcell.swc')
```

To manually create a morphology from a list of points in a similar format to SWC files, see *Morphology.from_points*.

Morphologies that are created in such a way will use standard names for the sections that allow for the short syntax shown in the previous sections: if a section has one or two child sections, then they will be called `L` and `R`, otherwise they will be numbered starting at `1`.

Morphologies with coordinates can also be created section by section, following the same syntax as for "schematic" morphologies:

```
soma = Soma(diameter=30*um, x=50*um, y=20*um)
cylinder = Cylinder(n=10, x=[0, 100]*um, diameter=1*um)
section = Section(n=5,
                  x=[0, 10, 20, 30, 40, 50]*um,
                  y=[0, 10, 20, 30, 40, 50]*um,
                  z=[0, 10, 10, 10, 10, 10]*um,
                  diameter=[6, 5, 4, 3, 2, 1]*um)
```

Note that the x, y, z attributes of *Morphology* and *SpatialNeuron* will return the coordinates at the midpoint of each compartment (as for all other attributes that vary over the length of a compartment, e.g. `diameter` or dis-

---

tance), but during construction the coordinates refer to the start and end of the section (`Cylinder`), respectively to the coordinates of the nodes between the compartments (`Section`).

A few additional remarks:

1. In the majority of simulations, coordinates are not used in the neuronal equations, therefore the coordinates are purely for visualization purposes and do not affect the simulation results in any way.

2. Coordinate specification cannot be combined with length specification – lengths are automatically calculated from the coordinates.

3. The coordinate specification can also be 1- or 2-dimensional (as in the first two examples above), the unspecified coordinate will use 0 μm.

4. All coordinates are interpreted relative to the parent compartment, i.e. the point (0 μm, 0 μm, 0 μm) refers to the end point of the previous compartment. Most of the time, the first element of the coordinate specification is therefore 0 μm, to continue a section where the previous one ended. However, it can be convenient to use a value different from 0 μm for sections connecting to the `Soma` to make them (visually) connect to a point on the sphere surface instead of the center of the sphere.

3.11.2 Creating a spatially extended neuron

A `SpatialNeuron` is a spatially extended neuron. It is created by specifying the morphology as a `Morphology` object, the equations for transmembrane currents, and optionally the specific membrane capacitance Cm and intracellular resistivity Ri:

```
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im=gL * (EL - v) : amp/meter**2
I : amp (point current)
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2, Ri=100*ohm*cm)
neuron.v = EL + 10*mV
```

Several state variables are created automatically: the `SpatialNeuron` inherits all the geometrical variables of the compartments (`length`, `diameter`, `area`, `volume`), as well as the `distance` variable that gives the distance to the soma. For morphologies that use coordinates, the x, y and z variables are provided as well. Additionally, a state variable Cm is created. It is initialized with the value given at construction, but it can be modified on a compartment per compartment basis (which is useful to model myelinated axons). The membrane potential is stored in state variable v.

Note that for all variable values that vary across a compartment (e.g. `distance`, x, y, z, v), the value that is reported is the value at the midpoint of the compartment.

The key state variable, which must be specified at construction, is Im. It is the total transmembrane current, expressed in units of current per area. This is a mandatory line in the definition of the model. The rest of the string description may include other state variables (differential equations or subexpressions) or parameters, exactly as in `NeuronGroup`. At every timestep, Brian integrates the state variables, calculates the transmembrane current at every point on the neuronal morphology, and updates v using the transmembrane current and the diffusion current, which is calculated based on the morphology and the intracellular resistivity. Note that the transmembrane current is a surfacic current, not the total current in the compartment. This choice means that the model equations are independent of the number of compartments chosen for the simulation. The space and time constants can obtained for any point of the neuron with the `space_constant` respectively `time_constant` attributes:

```
l = neuron.space_constant[0]
tau = neuron.time_constant[0]
```

The calculation is based on the local total conductance (not just the leak conductance), therefore, it can potentially vary during a simulation (e.g. decrease during an action potential). The reported value is only correct for compartments with a cylindrical geometry, though, it does not give reasonable values for compartments with strongly varying diameter.

To inject a current `I` at a particular point (e.g. through an electrode or a synapse), this current must be divided by the area of the compartment when inserted in the transmembrane current equation. This is done automatically when the flag `point current` is specified, as in the example above. This flag can apply only to subexpressions or parameters with amp units. Internally, the expression of the transmembrane current `Im` is simply augmented with `+I/area`. A current can then be injected in the first compartment of the neuron (generally the soma) as follows:

```
neuron.I[0] = 1*nA
```

State variables of the *SpatialNeuron* include all the compartments of that neuron (including subtrees). Therefore, the statement `neuron.v = EL + 10*mV` sets the membrane potential of the entire neuron at -60 mV.

Subtrees can be accessed by attribute (in the same way as in *Morphology* objects):

```
neuron.axon.gNa = 10*gL
```

Note that the state variables correspond to the entire subtree, not just the main section. That is, if the axon had branches, then the above statement would change `gNa` on the main section and all the sections in the subtree. To access the main section only, use the attribute `main`:

```
neuron.axon.main.gNa = 10*gL
```

A typical use case is when one wants to change parameter values at the soma only. For example, inserting an electrode current at the soma is done as follows:

```
neuron.main.I = 1*nA
```

A part of a section can be accessed as follows:

```
initial_segment = neuron.axon[10*um:50*um]
```

Finally, similar to the way that you can refer to a subset of neurons of a *NeuronGroup*, you can also index the *SpatialNeuron* object itself, e.g. to get a group representing only the first compartment of a cell (typically the soma), you can use:

```
soma = neuron[0]
```

In the same way as for sections, you can also use slices, either with the indices of compartments, or with the distance from the root:

```
first_compartments = neuron[:3]
first_compartments = neuron[0*um:30*um]
```

However, note that this is restricted to contiguous indices which most of the time means that all compartments indexed in this way have to be part of the same section. Such indices can be acquired directly from the morphology:

```
axon = neuron[morpho.axon.indices[:]]
```

or, more concisely:

```
axon = neuron[morpho.axon]
```

Synaptic inputs

There are two methods to have synapses on *SpatialNeuron*. The first one to insert synaptic equations directly in the neuron equations:

```
eqs='''
Im = gL * (EL - v) : amp/meter**2
Is = gs * (Es - v) : amp (point current)
dgs/dt = -gs/taus : siemens
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2, Ri=100*ohm*cm)
```

Note that, as for electrode stimulation, the synaptic current must be defined as a point current. Then we use a *Synapses* object to connect a spike source to the neuron:

```
S = Synapses(stimulation, neuron, on_pre='gs += w')
S.connect(i=0, j=50)
S.connect(i=1, j=100)
```

This creates two synapses, on compartments 50 and 100. One can specify the compartment number with its spatial position by indexing the morphology:

```
S.connect(i=0, j=morpho[25*um])
S.connect(i=1, j=morpho.axon[30*um])
```

In this method for creating synapses, there is a single value for the synaptic conductance in any compartment. This means that it will fail if there are several synapses onto the same compartment and synaptic equations are nonlinear. The second method, which works in such cases, is to have synaptic equations in the *Synapses* object:

```
eqs='''
Im = gL * (EL - v) : amp/meter**2
Is = gs * (Es - v) : amp (point current)
gs : siemens
'''
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1 * uF / cm ** 2, Ri=100 *↵
↪ohm * cm)
S = Synapses(stimulation, neuron, model='''dg/dt = -g/taus : siemens
                                            gs_post = g : siemens (summed)''',
             on_pre='g += w')
```

Here each synapse (instead of each compartment) has an associated value `g`, and all values of `g` for each compartment (i.e., all synapses targeting that compartment) are collected into the compartmental variable `gs`.

Detecting spikes

To detect and record spikes, we must specify a threshold condition, essentially in the same way as for a *NeuronGroup*:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold='v > 0*mV', refractory=↵
↪'v > -10*mV')
```

Here spikes are detected when the membrane potential v reaches 0 mV. Because there is generally no explicit reset in this type of model (although it is possible to specify one), v remains above 0 mV for some time. To avoid detecting spikes during this entire time, we specify a refractory period. In this case no spike is detected as long as v is greater than -10 mV. Another possibility could be:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold='m > 0.5', refractory=
↪'m > 0.4')
```

where m is the state variable for sodium channel activation (assuming this has been defined in the model). Here a spike is
detected when half of the sodium channels are open.

With the syntax above, spikes are detected in all compartments of the neuron. To detect them in a single compartment,
use the threshold_location keyword:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold='m > 0.5', threshold_
↪location=30,
                       refractory='m > 0.4')
```

In this case, spikes are only detecting in compartment number 30. Reset then applies locally to that compartment (if a
reset statement is defined). Again the location of the threshold can be specified with spatial position:

```
neuron = SpatialNeuron(morphology=morpho, model=eqs, threshold='m > 0.5',
                       threshold_location=morpho.axon[30*um],
                       refractory='m > 0.4')
```

Subgroups

In the same way that you can refer to a subset of neurons in a *NeuronGroup*, you can also refer to a subset of compart-
ments in a *SpatialNeuron*

3.12 Computational methods and efficiency

- *Runtime code generation*
- *Standalone code generation*
- *Compiler settings*

Brian has several different methods for running the computations in a simulation. The default mode is *Runtime code
generation*, which runs the simulation loop in Python but compiles and executes the modules doing the actual simulation
work (numerical integration, synaptic propagation, etc.) in a defined target language. Brian will select the best available
target language automatically. On Windows, to ensure that you get the advantages of compiled code, read the instructions
on installing a suitable compiler in *Requirements for C++ code generation*. Runtime mode has the advantage that you can
combine the computations performed by Brian with arbitrary Python code specified as *NetworkOperation*.

The fact that the simulation is run in Python means that there is a (potentially big) overhead for each simulated time step.
An alternative is to run Brian in with *Standalone code generation* – this is in general faster (for certain types of simulations
much faster) but cannot be used for all kinds of simulations. To enable this mode, add the following line after your Brian
import, but before your simulation code:

```
set_device('cpp_standalone')
```

For detailed control over the compilation process (both for runtime and standalone code generation), you can change the
Cleaning up after a run that are used.

___

The following topics are not essential for beginners.

___

3.12.1 Runtime code generation

Code generation means that Brian takes the Python code and strings in your model and generates code in one of several possible different languages which is then executed. The target language for this code generation process is set in the *codegen.target* preference. By default, this preference is set to `'auto'`, meaning that it will choose the compiled language target if possible and fall back to Python otherwise (also raising a warning). The compiled language target is `'cython'` which needs the Cython package in addition to a working C++ compiler. If you want to chose a code generation target explicitly (e.g. because you want to get rid of the warning that only the Python fallback is available), set the preference to `'numpy'` or `'cython'` at the beginning of your script:

```python
from brian2 import *
prefs.codegen.target = 'numpy'  # use the Python fallback
```

See *Preferences* for different ways of setting preferences.

Caching

When you run code with `cython` for the first time, it will take some time to compile the code. For short simulations, this can make these targets to appear slow compared to the `numpy` target where such compilation is not necessary. However, the compiled code is stored on disk and will be re-used for later runs, making these simulations start faster. If you run many simulations with different code (e.g. Brian's *test suite*), this code can take quite a bit of space on the disk. During the import of the `brian2` package, we check whether the size of the disk cache exceeds the value set by the *codegen.max_cache_dir_size* preference (by default, 1GB) and display a message if this is the case. You can clear the disk cache manually, or use the `clear_cache` function, e.g. `clear_cache('cython')`.

Note: If you run simulations on parallel on a machine using the Network File System, see *this known issue*.

3.12.2 Standalone code generation

Brian supports generating standalone code for multiple devices. In this mode, running a Brian script generates source code in a project tree for the target device/language. This code can then be compiled and run on the device, and modified if needed. At the moment, the only "device" supported is standalone C++ code. In some cases, the speed gains can be impressive, in particular for smaller networks with complicated spike propagation rules (such as STDP).

To use the C++ standalone mode, you only have to make very small changes to your script. The exact change depends on whether your script has only a single *run()* (or *Network.run*) call, or several of them:

Single run call

At the beginning of the script, i.e. after the import statements, add:

```
set_device('cpp_standalone')
```

The `CPPStandaloneDevice.build` function will be automatically called with default arguments right after the *run()* call. If you need non-standard arguments then you can specify them as part of the *set_device()* call:

```
set_device('cpp_standalone', directory='my_directory', debug=True)
```

Multiple run calls

At the beginning of the script, i.e. after the import statements, add:

```
set_device('cpp_standalone', build_on_run=False)
```

After the last *run()* call, call `device.build()` explicitly:

```
device.build(directory='output', compile=True, run=True, debug=False)
```

The `build` function has several arguments to specify the output directory, whether or not to compile and run the project after creating it and whether or not to compile it with debugging support or not.

Multiple builds

To run multiple full simulations (i.e. multiple `device.build` calls, not just multiple *run()* calls as discussed above), you have to reinitialize the device again:

```
device.reinit()
device.activate()
```

Note that the device "forgets" about all previously set build options provided to *set_device()* (most importantly the `build_on_run` option, but also e.g. the directory), you'll have to specify them as part of the `Device.activate` call. Also, `Device.activate` will reset the *defaultclock*, you'll therefore have to set its `dt` *after* the `activate` call if you want to use a non-default value.

Limitations

Not all features of Brian will work with C++ standalone, in particular Python based network operations and some array based syntax such as `S.w[0, :] = ...` will not work. If possible, rewrite these using string based syntax and they should work. Also note that since the Python code actually runs as normal, code that does something like this may not behave as you would like:

```
results = []
for val in vals:
    # set up a network
    run()
    results.append(result)
```

The current C++ standalone code generation only works for a fixed number of `run` statements, not with loops. If you need to do loops or other features not supported automatically, you can do so by inspecting the generated C++ source code and modifying it, or by inserting code directly into the main loop as follows:

```
device.insert_code('main', '''
cout << "Testing direct insertion of code." << endl;
''')
```

Variables

After a simulation has been run (after the *run()* call if *set_device()* has been called with `build_on_run` set to `True` or after the `Device.build` call with `run` set to `True`), state variables and monitored variables can be accessed using standard syntax, with a few exceptions (e.g. string expressions for indexing).

Multi-threading with OpenMP

> **Warning:** OpenMP code has not yet been well tested and so may be inaccurate.

When using the C++ standalone mode, you have the opportunity to turn on multi-threading, if your C++ compiler is compatible with OpenMP. By default, this option is turned off and only one thread is used. However, by changing the preferences of the codegen.cpp_standalone object, you can turn it on. To do so, just add the following line in your python script:

```
prefs.devices.cpp_standalone.openmp_threads = XX
```

XX should be a positive value representing the number of threads that will be used during the simulation. Note that the speedup will strongly depend on the network, so there is no guarantee that the speedup will be linear as a function of the number of threads. However, this is working fine for networks with not too small timestep (dt > 0.1ms), and results do not depend on the number of threads used in the simulation.

Customizing the build process

In standalone mode, a standard "make file" is used to orchestrate the compilation and linking. To provide additional arguments to the `make` command (respectively `nmake` on Windows), you can use the *devices.cpp_standalone.extra_make_args_unix* or *devices.cpp_standalone.extra_make_args_windows* preference. On Linux, this preference is by default set to `['-j']` to enable parallel compilation. Note that you can also use these arguments to overwrite variables in the make file, e.g. to use clang instead of the default gcc compiler:

```
prefs.devices.cpp_standalone.extra_make_args_unix += ['CC=clang++']
```

Cleaning up after a run

Standalone simulations store all results of a simulation (final state variable values and values stored in monitors) to disk. These results can take up quite significant amount of space, and you might therefore want to delete these results when you do not need them anymore. You can do this by using the device's *delete* method:

```
device.delete()
```

Be aware that deleting the data will make all access to state variables fail, including the access to values in monitors. You should therefore only delete the data after doing all analysis/plotting that you are interested in.

By default, this function will delete both the generated code and the data, i.e. the full project directory. If you want to keep the code (which typically takes up little space compared to the results), exclude it from the deletion:

```
device.delete(code=False)
```

If you added any additional files to the project directory manually, these will not be deleted by default. To delete the full directory regardless of its content, use the `force` option:

```
device.delete(force=True)
```

Note: When you initialize state variables with concrete values (and not with a string expression), they will be stored to disk from your Python script and loaded from disk at the beginning of the standalone run. Since these values are necessary for the compiled binary file to run, they are considered "code" from the point of view of the `delete` function.

3.12.3 Compiler settings

If using C++ code generation (either via cython or standalone), the compiler settings can make a big difference for the speed of the simulation. By default, Brian uses a set of compiler settings that switches on various optimizations and compiles for running on the same architecture where the code is compiled. This allows the compiler to make use of as many advanced instructions as possible, but reduces portability of the generated executable (which is not usually an issue).

If there are any issues with these compiler settings, for example because you are using an older version of the C++ compiler or because you want to run the generated code on a different architecture, you can change the settings by manually specifying the *codegen.cpp.extra_compile_args* preference (or by using *codegen.cpp.extra_compile_args_gcc* or *codegen.cpp.extra_compile_args_msvc* if you want to specify the settings for either compiler only).

3.13 Converting from integrated form to ODEs

Brian requires models to be expressed as systems of first order ordinary differential equations, and the effect of spikes to be expressed as (possibly delayed) one-off changes. However, many neuron models are given in *integrated form*. For example, one form of the Spike Response Model (SRM; Gerstner and Kistler 2002) is defined as

$$V(t) = \sum_i w_i \sum_{t_i} \text{PSP}(t - t_i) + V_{\text{rest}}$$

where $V(t)$ is the membrane potential, V_{rest} is the rest potential, w_i is the synaptic weight of synapse i, and t_i are the timings of the spikes coming from synapse i, and PSP is a postsynaptic potential function.

An example PSP is the α-function $\text{PSP}(t) = (t/\tau)e^{-t/\tau}$. For this function, we could rewrite the equation above in the following ODE form:

$$\tau \frac{dV}{dt} = V_{\text{rest}} - V + g$$
$$\tau \frac{dg}{dt} = -g$$
$$g \leftarrow g + w_i \quad \text{upon spike from synapse } i$$

This could then be written in Brian as:

```
eqs = '''
dV/dt = (V_rest-V+g)/tau : 1
dg/dt = -g/tau : 1
'''
G = NeuronGroup(N, eqs, ...)
```

```
...
S = Synapses(G, G, 'w : 1', on_pre='g += w')
```

To see that these two formulations are the same, you first solve the problem for the case of a single synapse and a single spike at time 0. The initial conditions at $t = 0$ will be $V(0) = V_{\text{rest}}$, $g(0) = w$.

To solve these equations, let's substitute $s = t/\tau$ and take derivatives with respect to s instead of t, set $u = V - V_{\text{rest}}$, and assume $w = 1$. This gives us the equations $u' = g - u$, $g' = -g$ with initial conditions $u(0) = 0$, $g(0) = 1$. At this point, you can either consult a textbook on solving linear systems of differential equations, or just plug this into Wolfram Alpha to get the solution $g(s) = e^{-s}$, $u(s) = se^{-s}$ which is equal to the PSP given above.

Now we use the linearity of these differential equations to see that it also works when $w \neq 0$ and for summing over multiple spikes at different times.

In general, to convert from integrated form to ODE form, see Köhn and Wörgötter (1998), Sánchez-Montañás (2001), and Jahnke et al. (1999). However, for some simple and widely used types of synapses, use the list below. In this list, we assume synapses are postsynaptic potentials, but you can replace $V(t)$ with a current or conductance for postsynaptic currents or conductances. In each case, we give the Brian code with unitless variables, where `eqs` is the differential equations for the target `NeuronGroup`, and `on_pre` is the argument to `Synapses`.

Exponential synapse $V(t) = e^{-t/\tau}$:

```
eqs = '''
dV/dt = -V/tau : 1
'''
on_pre = 'V += w'
```

Alpha synapse $V(t) = (t/\tau)e^{-t/\tau}$:

```
eqs = '''
dV/dt = (x-V)/tau : 1
dx/dt = -x/tau    : 1
'''
on_pre = 'x += w'
```

$V(t)$ reaches a maximum value of w/e at time $t = \tau$.

Biexponential synapse $V(t) = \frac{\tau_2}{\tau_2 - \tau_1}\left(e^{-t/\tau_1} - e^{-t/\tau_2}\right)$:

```
eqs = '''
dV/dt = ((tau_2 / tau_1) ** (tau_1 / (tau_2 - tau_1))*x-V)/tau_1 : 1
dx/dt = -x/tau_2                                                 : 1
'''
on_pre = 'x += w'
```

$V(t)$ reaches a maximum value of w at time $t = \frac{\tau_1 \tau_2}{\tau_2 - \tau_1} \log\left(\frac{\tau_2}{\tau_1}\right)$.

STDP

The weight update equation of the standard STDP is also often stated in an integrated form and can be converted to an ODE form. This is covered in *Tutorial 2*.

3.14 How to plot functions

Models of synapses and neurons are typically composed of a series of functions. To affirm their correct implementation a plot is often helpful.

Consider the following membrane voltage dependent Hodgkin-Huxley equations:

```python
from brian2 import *

VT = -63*mV

eq = Equations("""
alpha_m = 0.32*(mV**-1)*4*mV/exprel((13*mV-v+VT)/(4*mV))/ms : Hz
beta_m = 0.28*(mV**-1)*5*mV/exprel((v-VT-40*mV)/(5*mV))/ms : Hz
alpha_h = 0.128*exp((17*mV-v+VT)/(18*mV))/ms : Hz
beta_h = 4./(1+exp((40*mV-v+VT)/(5*mV)))/ms : Hz
alpha_n = 0.032*(mV**-1)*5*mV/exprel((15*mV-v+VT)/(5*mV))/ms : Hz
beta_n = .5*exp((10*mV-v+VT)/(40*mV))/ms : Hz
tau_n = 1/(alpha_n + beta_n) : second
tau_m = 1/(alpha_m + beta_m) : second
tau_h = 1/(alpha_h + beta_h) : second
""")
```

We can do the following to plot them as function of membrane voltage:

```python
group = NeuronGroup(100, eq + Equations("v : volt"))
group.v = np.linspace(-100, 100, len(group))*mV

plt.plot(group.v/mV, group.tau_m[:]/ms, label="tau_m")
plt.plot(group.v/mV, group.tau_n[:]/ms, label="tau_n")
plt.plot(group.v/mV, group.tau_h[:]/ms, label="tau_h")
plt.xlabel('membrane voltage / mV')
plt.ylabel('tau / ms')
plt.legend()
```
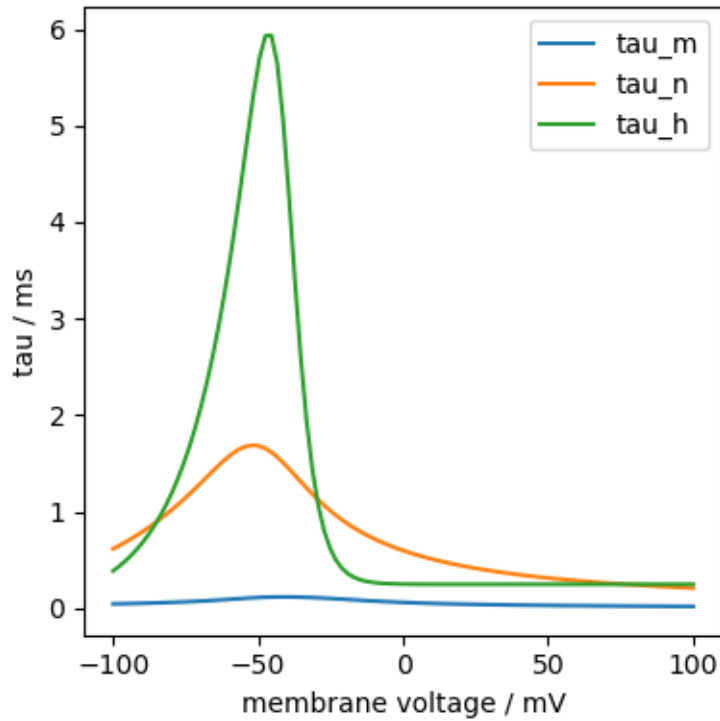
Note that we need to use `[:]` for the `tau_...` equations, because Brian cannot resolve the external constant `VT` otherwise. Alternatively we could have supplied the constant in the namespace of the *NeuronGroup*, see *Namespaces*.

ADVANCED GUIDE

This section has additional information on details not covered in the *User's guide*.

4.1 Functions

- *Default functions*
- *User-provided functions*
    - *Python code generation*
    - *Other code generation targets*
    - *Dependencies between functions*
    - *Additional compiler arguments*
    - *Arrays vs. scalar values in user-provided functions*
    - *Functions with context-dependent return values*
    - *Additional namespace*
    - *Data types*
    - *External source files*

All equations, expressions and statements in Brian can make use of mathematical functions. However, functions have to be prepared for use with Brian for two reasons: 1) Brian is strict about checking the consistency of units, therefore every function has to specify how it deals with units; 2) functions need to be implemented differently for different code generation targets.

Brian provides a number of default functions that are already prepared for use with numpy and C++ and also provides a mechanism for preparing new functions for use (see below).

4.1.1 Default functions

The following functions (stored in the `DEFAULT_FUNCTIONS` dictionary) are ready for use:

- Random numbers: `rand` (random numbers drawn from a uniform distribution between 0 and 1), `randn` (random numbers drawn from the standard normal distribution, i.e. with mean 0 and standard deviation 1), and `poisson` (discrete random numbers from a Poisson distribution with rate parameter λ)

- Elementary functions: `sqrt`, `exp`, `log`, `log10`, `abs`, `sign`

- Trigonometric functions: `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `arcsin`, `arccos`, `arctan`

- Functions for improved numerical accuracy: `expm1` (calculates `exp(x) - 1`, more accurate for x close to 0), `log1p` (calculates `log(1 + x)`, more accurate for x close to 0), and `exprel` (calculates `(exp(x) - 1)/x`, more accurate for x close to 0, and returning 1.0 instead of `NaN` for `x == 0`

- General utility functions: `clip`, `floor`, `ceil`

Brian also provides a special purpose function `int`, which can be used to convert an expression or variable into an integer value. This is especially useful for boolean values (which will be converted into 0 or 1), for example to have a conditional evaluation as part of an equation or statement which sometimes allows to circumvent the lack of an `if` statement. For example, the following reset statement resets the variable `v` to either `v_r1` or `v_r2`, depending on the value of `w`: `'v = v_r1 * int(w <= 0.5) + v_r2 * int(w > 0.5)'`

Finally, the function `timestep` is a function that takes a time and the length of a time step as an input and returns an integer corresponding to the respective time step. The advantage of using this function over a simple division is that it slightly shifts the time before dividing to avoid floating point issues. This function is used as part of the *Refractoriness* mechanism.

4.1.2 User-provided functions

Python code generation

If a function is only used in contexts that use Python code generation, preparing a function for use with Brian only means specifying its units. The simplest way to do this is to use the `check_units()` decorator:

```python
@check_units(x1=meter, y1=meter, x2=meter, y2=meter, result=meter)
def distance(x1, y1, x2, y2):
    return sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

Another option is to wrap the function in a `Function` object:

```python
def distance(x1, y1, x2, y2):
    return sqrt((x1 - x2)**2 + (y1 - y2)**2)
# wrap the distance function
distance = Function(distance, arg_units=[meter, meter, meter, meter],
                    return_unit=meter)
```

The use of Brian's unit system has the benefit of checking the consistency of units for every operation but at the expense of performance. Consider the following function, for example:

```python
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

When Brian runs a simulation, the state variables are stored and passed around without units for performance reasons. If the above function is used, however, Brian adds units to its input argument so that the operations inside the function do not fail with dimension mismatches. Accordingly, units are removed from the return value so that the function output

can be used with the rest of the code. For better performance, Brian can alter the namespace of the function when it is executed as part of the simulation and remove all the units, then pass values without units to the function. In the above example, this means making the symbol `nA` refer to `1e-9` and `Hz` to `1`. To use this mechanism, add the decorator `implementation()` with the `discard_units` keyword:

```python
@implementation('numpy', discard_units=True)
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Note that the use of the function *outside of simulation runs* is not affected, i.e. using `piecewise_linear` still requires a current in Ampere and returns a rate in Hertz. The `discard_units` mechanism does not work in all cases, e.g. it does not work if the function refers to units as `brian2.nA` instead of `nA`, if it uses imports inside the function (e.g. `from brian2 import nA`), etc. The `discard_units` can also be switched on for all functions without having to use the `implementation()` decorator by setting the *codegen.runtime.numpy.discard_units* preference.

Other code generation targets

To make a function available for other code generation targets (e.g. C++), implementations for these targets have to be added. This can be achieved using the `implementation()` decorator. The form of the code (e.g. a simple string or a dictionary of strings) necessary is target-dependent, for C++ both options are allowed, a simple string will be interpreted as filling the `'support_code'` block. Note that `'cpp'` is used to provide C++ implementations. An implementation for the C++ target could look like this:

```cpp
@implementation('cpp', '''
    double piecewise_linear(double I) {
        if (I < 1e-9)
            return 0;
        if (I > 3e-9)
            return 100;
        return (I/1e-9 - 1) * 50;
    }
    ''')
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Alternatively, `FunctionImplementation` objects can be added to the `Function` object.

The same sort of approach as for C++ works for Cython using the `'cython'` target. The example above would look like this:

```cython
@implementation('cython', '''
    cdef double piecewise_linear(double I):
        if I<1e-9:
            return 0.0
        elif I>3e-9:
            return 100.0
        return (I/1e-9-1)*50
    ''')
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

Dependencies between functions

The code generation mechanism for user-defined functions only adds the source code for a function when it is necessary. If a user-defined function refers to another function in its source code, it therefore has to explicitly state this dependency so that the code of the dependency is added as well:

```
@implementation('cpp','''
    double rectified_linear(double x)
    {
        return clip(x, 0, INFINITY);
    }''',
    dependencies={'clip': DEFAULT_FUNCTIONS['clip']}
    )
@check_units(x=1, result=1)
def rectified_linear(x):
    return np.clip(x, 0, np.inf)
```

Note: The dependency mechanism is unnecessary for the `numpy` code generation target, since functions are defined as actual Python functions and not as code given in a string.

Additional compiler arguments

If the code for a function needs additional compiler options to work, e.g. to link to an external library, these options can be provided as keyword arguments to the `@implementation` decorator. E.g. to link C++ code to the `foo` library which is stored in the directory `/usr/local/foo`, use:

```
@implementation('cpp', '...',
 libraries=['foo'], library_dirs=['/usr/local/foo'])
```

These arguments can also be used to refer to external source files, see *below*. Equivalent arguments can also be set as global *Preferences* in which case they apply to all code and not only to code referring to the respective function. Note that in C++ standalone mode, all files are compiled together, and therefore the additional compiler arguments provided to functions are always combined with the preferences into a common set of settings that is applied to all code.

The list of currently supported additional arguments (for further explications, see the respective *Preferences* and the Python documentation of the `distutils.core.Extension` class):

keyword	C++ standalone	Cython
`headers`	✓	✗
`sources`	✓	✓
`define_macros`	✓	✗
`libraries`	✓	✓
`include_dirs`	✓	✓
`library_dirs`	✓	✓
`runtime_library_dirs`	✓	✓

Arrays vs. scalar values in user-provided functions

Equations, expressions and abstract code statements are always implicitly referring to all the neurons in a `NeuronGroup`, all the synapses in a `Synapses` object, etc. Therefore, function calls also apply to more than a single value. The way in which this is handled differs between code generation targets that support vectorized expressions (e.g. the `numpy` target) and targets that don't (e.g. the `cpp_standalone` mode). If the code generation target supports vectorized expressions, it will receive an array of values. For example, in the `piecewise_linear` example above, the argument `I` will be an array of values and the function returns an array of values. For code generation without support for vectorized expressions, all code will be executed in a loop (over neurons, over synapses, …), the function will therefore be called several times with a single value each time.

In both cases, the function will only receive the "relevant" values, meaning that if for example a function is evaluated as part of a reset statement, it will only receive values for the neurons that just spiked.

Functions with context-dependent return values

When using the `numpy` target, functions have to return an array of values (e.g. one value for each neuron). In some cases, the number of values to return cannot be deduced from the function's arguments. Most importantly, this is the case for random numbers: a call to `rand()` has to return one value for each neuron if it is part of a neuron's equations, but only one value for each neuron that spiked during the time step if it is part of the reset statement. Such function are said to "auto vectorise", which means that their implementation receives an additional array argument `_vectorisation_idx`; the length of this array determines the number of values the function should return. This argument is also provided to functions for other code generation targets, but in these cases it is a single value (e.g. the index of the neuron), and is currently ignored. To enable this property on a user-defined function, you'll currently have to manually create a `Function` object:

```python
def exponential_rand(l, _vectorisation_idx):
    '''Generate a number from an exponential distribution using inverse
       transform sampling'''
    uniform = np.random.rand(len(_vectorisation_idx))
    return -(1/l)*np.log(1 - uniform)

exponential_rand = Function(exponential_rand, arg_units=[1], return_unit=1,
                            stateless=False, auto_vectorise=True)
```

Implementations for other code generation targets can then be added using the `add_implementation` mechanism:

```python
cpp_code = '''
double exponential_rand(double l, int _vectorisation_idx)
{
    double uniform = rand(_vectorisation_idx);
    return -(1/l)*log(1 - uniform);
}
'''
exponential_rand.implementations.add_implementation('cpp', cpp_code,
                                                    dependencies={'rand': DEFAULT_
→FUNCTIONS['rand'],
                                                                  'log': DEFAULT_
→FUNCTIONS['log']})
```

Note that by referring to the `rand` function, the new random number generator will automatically generate reproducible random numbers if the `seed()` function is use to set its seed. Restoring the random number state with `restore()` will have the expected effect as well.

Additional namespace

Some functions need additional data to compute a result, e.g. a *TimedArray* needs access to the underlying array. For the `numpy` target, a function can simply use a reference to an object defined outside the function, there is no need to explicitly pass values in a namespace. For the other code language targets, values can be passed in the `namespace` argument of the *implementation()* decorator or the *add_implementation* method. The namespace values are then accessible in the function code under the given name, prefixed with _namespace. Note that this mechanism should only be used for numpy arrays or general objects (e.g. function references to call Python functions from Cython code). Scalar values should be directly included in the function code, by using a "dynamic implemention" (see *add_dynamic_implementation*).

See *TimedArray* and *BinomialFunction* for examples that use this mechanism.

Data types

By default, functions are assumed to take any type of argument, and return a floating point value. If you want to put a restriction on the type of an argument, or specify that the return type should be something other than float, either declare it as a *Function* (and see its documentation on specifying types) or use the *declare_types()* decorator, e.g.:

```
@check_units(a=1, b=1, result=1)
@declare_types(a='integer', result='highest')
def f(a, b):
    return a*b
```

This is potentially important if you have functions that return integer or boolean values, because Brian's code generation optimisation step will make some potentially incorrect simplifications if it assumes that the return type is floating point.

External source files

Code for functions can also be provided via external files in the target language. This can be especially useful for linking to existing code without having to include it a second time in the Python script. For C++-based code generation targets (i.e. the C++ standalone mode), the external code should be in a file that is provided as an argument to the `sources` keyword, together with a header file whose name is provided to `headers` (see the note for the *codegen.cpp.headers* preference about the necessary format). Since the main simulation code is compiled and executed in a different directory, you should also point the compiler towards the directory of the header file via the `include_dirs` keyword. For the same reason, use an absolute path for the source file. For example, the `piecewise_linear` function from above can be implemented with external files as follows:

```
//file: piecewise_linear.h
double piecewise_linear(double);
```

```
//file: piecewise_linear.cpp
double piecewise_linear(double I) {
    if (I < 1e-9)
        return 0;
    if (I > 3e-9)
        return 100;
    return (I/1e-9 - 1) * 50;
}
```

```
# Python script

# Get the absolute directory of this Python script, the C++ files are
```

(continues on next page)

```python
# expected to be stored alongside of it
import os
current_dir = os.path.abspath(os.path.dirname(__file__))


@implementation('cpp', '// all code in piecewise_linear.cpp',
                sources=[os.path.join(current_dir,
                                      'piecewise_linear.cpp')],
                headers=['"piecewise_linear.h"'],
                include_dirs=[current_dir])
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

For Cython, the process is very similar (see the Cython documentation for general information). The name of the header file does not need to be specified, it is expected to have the same name as the source file (except for the `.pxd` extension). The source and header files will be automatically copied to the cache directory where Cython files are compiled, they therefore have to be imported as top-level modules, regardless of whether the executed Python code is itself in a package or module.

A Cython equivalent of above's C++ example can be written as:

```python
# file: piecewise_linear.pxd
cdef double piecewise_linear(double)
```

```python
# file: piecewise_linear.pyx
cdef double piecewise_linear(double I):
    if I<1e-9:
        return 0.0
    elif I>3e-9:
        return 100.0
    return (I/1e-9-1)*50
```

```python
# Python script

# Get the absolute directory of this Python script, the Cython files
# are expected to be stored alongside of it
import os
current_dir = os.path.abspath(os.path.dirname(__file__))


@implementation('cython',
                'from piecewise_linear cimport piecewise_linear',
                sources=[os.path.join(current_dir,
                                      'piecewise_linear.pyx')])
@check_units(I=amp, result=Hz)
def piecewise_linear(I):
    return clip((I-1*nA) * 50*Hz/nA, 0*Hz, 100*Hz)
```

4.2 Preferences

Brian has a system of global preferences that affect how certain objects behave. These can be set either in scripts by using the *prefs* object or in a file. Each preference looks like codegen.cpp.compiler, i.e. dotted names.

4.2.1 Accessing and setting preferences

Preferences can be accessed and set either keyword-based or attribute-based. The following are equivalent:

```
prefs['codegen.cpp.compiler'] = 'unix'
prefs.codegen.cpp.compiler = 'unix'
```

Using the attribute-based form can be particulary useful for interactive work, e.g. in ipython, as it offers autocompletion and documentation. In ipython, prefs.codegen.cpp? would display a docstring with all the preferences available in the codegen.cpp category.

4.2.2 Preference files

Preferences are stored in a hierarchy of files, with the following order (each step overrides the values in the previous step but no error is raised if one is missing):

- The user default are stored in ~/.brian/user_preferences (which works on Windows as well as Linux). The ~ symbol refers to the user directory.
- The file brian_preferences in the current directory.

The preference files are of the following form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
b.e = 3
```

This would set preferences a.b.c=1, a.b.d=2 and a.b.e=3.

```
#---------------------------------------------------------------------------
# Logging system preferences
#---------------------------------------------------------------------------

[logging]

# What log level to use for the log written to the console.
#
# Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

console_log_level = 'INFO'

# Whether to delete the log and script file on exit.
#
# If set to ``True`` (the default), log files (and the copy of the main
# script) will be deleted after the brian process has exited, unless an
# uncaught exception occurred. If set to ``False``, all log files will be
# kept.
```

```
delete_log_on_exit = True


# Whether to display a text for uncaught errors, mentioning the location
# of the log file, the mailing list and the github issues.
#
# Defaults to ``True``.


display_brian_error_message = True


# Whether to log to a file or not.
#
# If set to ``True`` (the default), logging information will be written
# to a file. The log level can be set via the `logging.file_log_level`
# preference.


file_log = True


# What log level to use for the log written to the log file.
#
# In case file logging is activated (see `logging.file_log`), which log
# level should be used for logging. Has to be one of CRITICAL, ERROR,
# WARNING, INFO, DEBUG or DIAGNOSTIC.


file_log_level = 'DIAGNOSTIC'


# The maximum size for the debug log before it will be rotated.
#
# If set to any value ``> 0``, the debug log will be rotated once
# this size is reached. Rotating the log means that the old debug log
# will be moved into a file in the same directory but with suffix ``".1"``
# and the a new log file will be created with the same pathname as the
# original file. Only one backup is kept; if a file with suffix ``".1"``
# already exists when rotating, it will be overwritten.
# If set to ``0``, no log rotation will be applied.
# The default setting rotates the log file after 10MB.


file_log_max_size = 10000000


# Whether to save a copy of the script that is run.
#
# If set to ``True`` (the default), a copy of the currently run script
# is saved to a temporary location. It is deleted after a successful
# run (unless `logging.delete_log_on_exit` is ``False``) but is kept after
# an uncaught exception occured. This can be helpful for debugging,
# in particular when several simulations are running in parallel.


save_script = True


# Whether or not to redirect stdout/stderr to null at certain places.
#
# This silences a lot of annoying compiler output, but will also hide
# error messages making it harder to debug problems. You can always
# temporarily switch it off when debugging. If
# `logging.std_redirection_to_file` is set to ``True`` as well, then the
# output is saved to a file and if an error occurs the name of this file
# will be printed.
```

```
std_redirection = True

# Whether to redirect stdout/stderr to a file.
#
# If both ``logging.std_redirection`` and this preference are set to
# ``True``, all standard output/error (most importantly output from
# the compiler) will be stored in files and if an error occurs the name
# of this file will be printed. If `logging.std_redirection` is ``True``
# and this preference is ``False``, then all standard output/error will
# be completely suppressed, i.e. neither be displayed nor stored in a
# file.
#
# The value of this preference is ignore if `logging.std_redirection` is
# set to ``False``.

std_redirection_to_file = True


#-------------------------------------------------------------------------------
# Runtime codegen preferences (see subcategories for individual targets)
#-------------------------------------------------------------------------------

[codegen.runtime]


#-------------------------------------------------------------------------------
# Codegen generator preferences (see subcategories for individual languages)
#-------------------------------------------------------------------------------

[codegen.generators]


#-------------------------------------------------------------------------------
# C++ compilation preferences
#-------------------------------------------------------------------------------

[codegen.cpp]

# Compiler to use (uses default if empty).
# Should be ``'unix'`` or ``'msvc'``.
#
# To specify a specific compiler binary on unix systems, set the `CXX` environment
# variable instead.

compiler = ''

# List of macros to define; each macro is defined using a 2-tuple,
# where 'value' is either the string to define it to or None to
# define it without a particular value (equivalent of "#define
# FOO" in source or -DFOO on Unix C compiler command line).

define_macros = []

# Extra arguments to pass to compiler (if None, use either
# ``extra_compile_args_gcc`` or ``extra_compile_args_msvc``).

extra_compile_args = None

# Extra compile arguments to pass to GCC compiler
```

```
extra_compile_args_gcc = ['-w', '-O3', '-ffast-math', '-fno-finite-math-only', '-
↪march=native', '-std=c++11']

# Extra compile arguments to pass to MSVC compiler (the default
# ``/arch:`` flag is determined based on the processor architecture)

extra_compile_args_msvc = ['/Ox', '/w', '', '/MP']

# Any extra platform- and compiler-specific information to use when
# linking object files together.

extra_link_args = []

# A list of strings specifying header files to use when compiling the
# code. The list might look like ["<vector>","'my_header'"]. Note that
# the header strings need to be in a form than can be pasted at the end
# of a #include statement in the C++ code.

headers = []

# Include directories to use.
# The default value is ``$prefix/include`` (or ``$prefix/Library/include``
# on Windows), where ``$prefix`` is Python's site-specific directory
# prefix as returned by `sys.prefix`. This will make compilation use
# library files installed into a conda environment.

include_dirs = ['/path/to/your/Python/environment/include']

# List of library names (not filenames or paths) to link against.

libraries = []

# List of directories to search for C/C++ libraries at link time.
# The default value is ``$prefix/lib`` (or ``$prefix/Library/lib``
# on Windows), where ``$prefix`` is Python's site-specific directory
# prefix as returned by `sys.prefix`. This will make compilation use
# library files installed into a conda environment.

library_dirs = ['/path/to/your/Python/environment/lib']

# MSVC architecture name (or use system architectue by default).
#
# Could take values such as x86, amd64, etc.

msvc_architecture = ''

# Location of the MSVC command line tool (or search for best by default).

msvc_vars_location = ''

# List of directories to search for C/C++ libraries at run time.
# The default value is ``$prefix/lib`` (not used on Windows), where
# ``$prefix`` is Python's site-specific directory prefix as returned by
# `sys.prefix`. This will make compilation use library files installed
# into a conda environment.
```

```
runtime_library_dirs = ['/path/to/your/Python/environment/lib']


#-------------------------------------------------------------------------------
# C++ codegen preferences
#-------------------------------------------------------------------------------

[codegen.generators.cpp]

# Adds code to flush denormals to zero.
#
# The code is gcc and architecture specific, so may not compile on all
# platforms. The code, for reference is::
#
#     #define CSR_FLUSH_TO_ZERO         (1 << 15)
#     unsigned csr = __builtin_ia32_stmxcsr();
#     csr |= CSR_FLUSH_TO_ZERO;
#     __builtin_ia32_ldmxcsr(csr);
#
# Found at `<http://stackoverflow.com/questions/2487653/avoiding-denormal-values-in-c>
↪`_.

flush_denormals = False

# The keyword used for the given compiler to declare pointers as restricted.
#
# This keyword is different on different compilers, the default works for
# gcc and MSVS.

restrict_keyword = '__restrict'


#-------------------------------------------------------------------------------
# Device preferences
#-------------------------------------------------------------------------------

[devices]


#-------------------------------------------------------------------------------
# Directory containing GSL code
#-------------------------------------------------------------------------------

[GSL]

# Set path to directory containing GSL header files (gsl_odeiv2.h etc.)
# If this directory is already in Python's include (e.g. because of conda
↪installation), this path can be set to None.

directory = None


#-------------------------------------------------------------------------------
# Numpy runtime codegen preferences
#-------------------------------------------------------------------------------

[codegen.runtime.numpy]

# Whether to change the namespace of user-specifed functions to remove
# units.
```

```python
discard_units = False


#-------------------------------------------------------------------------------
# Cython runtime codegen preferences
#-------------------------------------------------------------------------------

[codegen.runtime.cython]

# Location of the cache directory for Cython files. By default,
# will be stored in a ``brian_extensions`` subdirectory
# where Cython inline stores its temporary files
# (the result of ``get_cython_cache_dir()``).

cache_dir = None

# Whether to delete source files after compiling. The Cython
# source files can take a significant amount of disk space, and
# are not used anymore when the compiled library file exists.
# They are therefore deleted by default, but keeping them around
# can be useful for debugging.

delete_source_files = True

# Whether to use a lock file to prevent simultaneous write access
# to cython .pyx and .so files.

multiprocess_safe = True


#-------------------------------------------------------------------------------
# Code generation preferences
#-------------------------------------------------------------------------------

[codegen]

# Whether to pull out scalar expressions out of the statements, so that
# they are only evaluated once instead of once for every neuron/synapse/...
# Can be switched off, e.g. because it complicates the code (and the same
# optimisation is already performed by the compiler) or because the
# code generation target does not deal well with it. Defaults to ``True``.

loop_invariant_optimisations = True

# The size of a directory (in MB) with cached code for Cython that triggers a warning.
# Set to 0 to never get a warning.

max_cache_dir_size = 1000

# Default target for the evaluation of string expressions (e.g. when
# indexing state variables). Should normally not be changed from the
# default numpy target, because the overhead of compiling code is not
# worth the speed gain for simple expressions.
#
# Accepts the same arguments as `codegen.target`, except for ``'auto'``

string_expression_target = 'numpy'

# Default target for code generation.
```

```
#
# Can be a string, in which case it should be one of:
#
# * ``'auto'`` the default, automatically chose the best code generation
#   target available.
# * ``'cython'``, uses the Cython package to generate C++ code. Needs a
#   working installation of Cython and a C++ compiler.
# * ``'numpy'`` works on all platforms and doesn't need a C compiler but
#   is often less efficient.
#
# Or it can be a ``CodeObject`` class.

target = 'auto'


#-------------------------------------------------------------------------------
# Network preferences
#-------------------------------------------------------------------------------

[core.network]

# Default schedule used for networks that
# don't specify a schedule.

default_schedule = ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']


#-------------------------------------------------------------------------------
# C++ standalone preferences
#-------------------------------------------------------------------------------

[devices.cpp_standalone]

# Additional flags to pass to the GNU make command on Linux/OS-X.
# Defaults to "-j" for parallel compilation.

extra_make_args_unix = ['-j']

# Additional flags to pass to the nmake command on Windows. By default, no
# additional flags are passed.

extra_make_args_windows = []

# The make command used to compile the standalone project. Defaults to the
# standard GNU make commane "make".

make_cmd_unix = 'make'

# DEPRECATED. Previously used to chose the strategy to parallelize the
# solution of the three tridiagonal systems for multicompartmental
# neurons. Now, its value is ignored.

openmp_spatialneuron_strategy = None

# The number of threads to use if OpenMP is turned on. By default, this value is set
# →to 0 and the C++ code
# is generated without any reference to OpenMP. If greater than 0, then the
# →corresponding number of threads
# are used to launch the simulation.
```

```
openmp_threads = 0

# The command used to run the compiled standalone project. Defaults to executing
# the compiled binary with "./main". Must be a single binary as string or a list
# of command arguments (e.g. ["./binary", "--key", "value"]).

run_cmd_unix = './main'

# Dictionary of environment variables and their values that will be set
# during the execution of the standalone code.

run_environment_variables = {'LD_BIND_NOW': '1'}

#-------------------------------------------------------------------------------
# Core Brian preferences
#-------------------------------------------------------------------------------

[core]

# Default dtype for all arrays of scalars (state variables, weights, etc.).

default_float_dtype = float64

# Default dtype for all arrays of integer scalars.

default_integer_dtype = int32

# Whether to raise an error for outdated dependencies (``True``) or just
# a warning (``False``).

outdated_dependency_error = True


#-------------------------------------------------------------------------------
# Preferences to enable legacy behaviour
#-------------------------------------------------------------------------------

[legacy]

# Whether to use the semantics for checking the refractoriness condition
# that were in place up until (including) version 2.1.2. In that
# implementation, refractory periods that were multiples of dt could lead
# to a varying number of refractory timesteps due to the nature of
# floating point comparisons). This preference is only provided for exact
# reproducibility of previously obtained results, new simulations should
# use the improved mechanism which uses a more robust mechanism to
# convert refractoriness into timesteps. Defaults to ``False``.

refractory_timing = False
```

4.2.3 List of preferences

Brian itself defines the following preferences (including their default values):

GSL

Directory containing GSL code

GSL.directory = None Set path to directory containing GSL header files (gsl_odeiv2.h etc.) If this directory is already in Python's include (e.g. because of conda installation), this path can be set to None.

codegen

Code generation preferences

`codegen.loop_invariant_optimisations = True`

Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/… Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.max_cache_dir_size = 1000`

The size of a directory (in MB) with cached code for Cython that triggers a warning. Set to 0 to never get a warning.

`codegen.string_expression_target = 'numpy'`

Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.
- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.
- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

codegen.cpp

C++ compilation preferences

`codegen.cpp.compiler = ''`

Compiler to use (uses default if empty). Should be `'unix'` or `'msvc'`.

To specify a specific compiler binary on unix systems, set the `CXX` environment variable instead.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args` = `None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvc`).

`codegen.cpp.extra_compile_args_gcc` = `['-w', '-O3', '-ffast-math', '-fno-finite-math-only', '-march=native', '-std=c++11']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc` = `['/Ox', '/w', '', '/MP']`

Extra compile arguments to pass to MSVC compiler (the default `/arch:` flag is determined based on the processor architecture)

`codegen.cpp.extra_link_args` = `[]`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers` = `[]`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header'"]. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs` = `['/path/to/your/Python/environment/include']`

Include directories to use. The default value is `$prefix/include` (or `$prefix/Library/include` on Windows), where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

`codegen.cpp.libraries` = `[]`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs` = `['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at link time. The default value is `$prefix/lib` (or `$prefix/Library/lib` on Windows), where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

`codegen.cpp.msvc_architecture` = `''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location` = `''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs` = `['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at run time. The default value is `$prefix/lib` (not used on Windows), where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

codegen.generators

Codegen generator preferences (see subcategories for individual languages)

codegen.generators.cpp

C++ codegen preferences

`codegen.generators.cpp.flush_denormals` = `False`

> Adds code to flush denormals to zero.
>
> The code is gcc and architecture specific, so may not compile on all platforms. The code, for reference is:

```
#define CSR_FLUSH_TO_ZERO         (1 << 15)
unsigned csr = __builtin_ia32_stmxcsr();
csr |= CSR_FLUSH_TO_ZERO;
__builtin_ia32_ldmxcsr(csr);
```

> Found at http://stackoverflow.com/questions/2487653/avoiding-denormal-values-in-c.

`codegen.generators.cpp.restrict_keyword` = `'__restrict'`

> The keyword used for the given compiler to declare pointers as restricted.
>
> This keyword is different on different compilers, the default works for gcc and MSVS.

codegen.runtime

Runtime codegen preferences (see subcategories for individual targets)

codegen.runtime.cython

Cython runtime codegen preferences

`codegen.runtime.cython.cache_dir` = `None`

> Location of the cache directory for Cython files. By default, will be stored in a `brian_extensions` subdirectory where Cython inline stores its temporary files (the result of `get_cython_cache_dir()`).

`codegen.runtime.cython.delete_source_files` = `True`

> Whether to delete source files after compiling. The Cython source files can take a significant amount of disk space, and are not used anymore when the compiled library file exists. They are therefore deleted by default, but keeping them around can be useful for debugging.

`codegen.runtime.cython.multiprocess_safe` = `True`

> Whether to use a lock file to prevent simultaneous write access to cython .pyx and .so files.

codegen.runtime.numpy

Numpy runtime codegen preferences

`codegen.runtime.numpy.discard_units` = `False`

> Whether to change the namespace of user-specifed functions to remove units.

core

Core Brian preferences

`core.default_float_dtype` = `float64`

> Default dtype for all arrays of scalars (state variables, weights, etc.).

`core.default_integer_dtype` = `int32`

> Default dtype for all arrays of integer scalars.

`core.outdated_dependency_error` = `True`

> Whether to raise an error for outdated dependencies (`True`) or just a warning (`False`).

core.network

Network preferences

```
core.network.default_schedule = ['start', 'groups', 'thresholds', 'synapses',
'resets', 'end']
```

> Default schedule used for networks that don't specify a schedule.

devices

Device preferences

devices.cpp_standalone

C++ standalone preferences

```
devices.cpp_standalone.extra_make_args_unix = ['-j']
```

> Additional flags to pass to the GNU make command on Linux/OS-X. Defaults to "-j" for parallel compilation.

```
devices.cpp_standalone.extra_make_args_windows = []
```

> Additional flags to pass to the nmake command on Windows. By default, no additional flags are passed.

```
devices.cpp_standalone.make_cmd_unix = 'make'
```

> The make command used to compile the standalone project. Defaults to the standard GNU make commane "make".

```
devices.cpp_standalone.openmp_spatialneuron_strategy = None
```

> DEPRECATED. Previously used to chose the strategy to parallelize the solution of the three tridiagonal systems for multicompartmental neurons. Now, its value is ignored.

```
devices.cpp_standalone.openmp_threads = 0
```

> The number of threads to use if OpenMP is turned on. By default, this value is set to 0 and the C++ code is generated without any reference to OpenMP. If greater than 0, then the corresponding number of threads are used to launch the simulation.

```
devices.cpp_standalone.run_cmd_unix = './main'
```

> The command used to run the compiled standalone project. Defaults to executing the compiled binary with "./main". Must be a single binary as string or a list of command arguments (e.g. ["./binary", "–key", "value"]).

```
devices.cpp_standalone.run_environment_variables = {'LD_BIND_NOW': '1'}
```

> Dictionary of environment variables and their values that will be set during the execution of the standalone code.

legacy

Preferences to enable legacy behaviour

```
legacy.refractory_timing = False
```

> Whether to use the semantics for checking the refractoriness condition that were in place up until (including) version 2.1.2. In that implementation, refractory periods that were multiples of dt could lead to a varying number of refractory timesteps due to the nature of floating point comparisons). This preference is only provided for exact reproducibility of previously obtained results, new simulations should use the improved mechanism which uses a more robust mechanism to convert refractoriness into timesteps. Defaults to `False`.

logging

Logging system preferences

`logging.console_log_level = 'INFO'`

> What log level to use for the log written to the console.
>
> Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.delete_log_on_exit = True`

> Whether to delete the log and script file on exit.
>
> If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occurred. If set to `False`, all log files will be kept.

`logging.display_brian_error_message = True`

> Whether to display a text for uncaught errors, mentioning the location of the log file, the mailing list and the github issues.
>
> Defaults to `True`.

`logging.file_log = True`

> Whether to log to a file or not.
>
> If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level = 'DIAGNOSTIC'`

> What log level to use for the log written to the log file.
>
> In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.file_log_max_size = 10000000`

> The maximum size for the debug log before it will be rotated.
>
> If set to any value > 0, the debug log will be rotated once this size is reached. Rotating the log means that the old debug log will be moved into a file in the same directory but with suffix `".1"` and the a new log file will be created with the same pathname as the original file. Only one backup is kept; if a file with suffix `".1"` already exists when rotating, it will be overwritten. If set to 0, no log rotation will be applied. The default setting rotates the log file after 10MB.

`logging.save_script = True`

> Whether to save a copy of the script that is run.
>
> If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection = True`

> Whether or not to redirect stdout/stderr to null at certain places.
>
> This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. If *logging.std_redirection_to_file* is set to `True` as well, then the output is saved to a file and if an error occurs the name of this file will be printed.

`logging.std_redirection_to_file = True`

Whether to redirect stdout/stderr to a file.

If both `logging.std_redirection` and this preference are set to `True`, all standard output/error (most importantly output from the compiler) will be stored in files and if an error occurs the name of this file will be printed. If *logging.std_redirection* is `True` and this preference is `False`, then all standard output/error will be completely suppressed, i.e. neither be displayed nor stored in a file.

The value of this preference is ignore if *logging.std_redirection* is set to `False`.

4.3 Logging

Brian uses a logging system to display warnings and general information messages to the user, as well as writing them to a file with more detailed information, useful for debugging. Each log message has one of the following "log levels":

ERROR Only used when an exception is raised, i.e. an error occurs and the current operation is interrupted. *Example:* You use a variable name in an equation that Brian does not recognize.

WARNING Brian thinks that something is most likely a bug, but it cannot be sure. *Example:* You use a *Synapses* object without any synapses in your simulation.

INFO Brian wants to make the user aware of some automatic choice that it did for the user. *Example:* You did not specify an integration `method` for a *NeuronGroup* and therefore Brian chose an appropriate method for you.

DEBUG Additional information that might be useful when a simulation is not working as expected. *Example:* The integration timestep used during the simulation.

DIAGNOSTIC Additional information useful when tracking down bugs in Brian itself. *Example:* The generated code for a `CodeObject`.

By default, all messages are written to the log file and all messages of level `INFO` and above are displayed on the console. To change what messages are displayed, see below.

Note: By default, the log file is deleted after a successful simulation run, i.e. when the simulation exited without an error. To keep the log around, set the *logging.delete_log_on_exit* preference to `False`.

4.3.1 Showing/hiding log messages

If you want to change what messages are displayed on the console, you can call a method of the method of *BrianLogger*:

```
BrianLogger.log_level_debug()  # now also display debug messages
```

It is also possible to suppress messages for certain sub-hierarchies by using *BrianLogger.suppress_hierarchy*:

```
# Suppress code generation messages on the console
BrianLogger.suppress_hierarchy('brian2.codegen')
# Suppress preference messages even in the log file
BrianLogger.suppress_hierarchy('brian2.core.preferences',
                               filter_log_file=True)
```

Similarly, messages ending in a certain name can be suppressed with *BrianLogger.suppress_name*:

```
# Suppress resolution conflict warnings
BrianLogger.suppress_name('resolution_conflict')
```

These functions should be used with care, as they suppresses messages independent of the level, i.e. even warning and error messages.

4.3.2 Preferences

You can also change details of the logging system via Brian's *Preferences* system. With this mechanism, you can switch the logging to a file off completely (by setting *logging.file_log* to `False`) or have it log less messages (by setting *logging.file_log_level* to a level higher than `DIAGNOSTIC`) – this can be important for long-running simulations where the log might otherwise take up a lot of disk space. For a list of all preferences related to logging, see the documentation of the `brian2.utils.logger` module.

> **Warning:** Most of the logging preferences are only taken into account during the initialization of the logging system which takes place as soon as `brian2` is imported. Therefore, if you use e.g. `prefs.logging.file_log = False` in your script, this will not have the intended effect! Instead, set these preferences in a file (see *Preferences*).

4.4 Namespaces

`Equations` can contain references to external parameters or functions. During the initialisation of a `NeuronGroup` or a `Synapses` object, this *namespace* can be provided as an argument. This is a group-specific namespace that will only be used for names in the context of the respective group. Note that units and a set of standard functions are always provided and should not be given explicitly. This namespace does not necessarily need to be exhaustive at the time of the creation of the `NeuronGroup`/`Synapses`, entries can be added (or modified) at a later stage via the `namespace` attribute (e.g. `G.namespace['tau'] = 10*ms`).

At the point of the call to the `Network.run` namespace, any group-specific namespace will be augmented by the "run namespace". This namespace can be either given explicitly as an argument to the `run` method or it will be taken from the locals and globals surrounding the call. A warning will be emitted if a name is defined in more than one namespace.

To summarize: an external identifier will be looked up in the context of an object such as `NeuronGroup` or `Synapses`. It will follow the following resolution hierarchy:

1. Default unit and function names.

2. Names defined in the explicit group-specific namespace.

3. Names in the run namespace which is either explicitly given or the implicit namespace surrounding the run call.

Note that if you completely specify your namespaces at the `Group` level, you should probably pass an empty dictionary as the namespace argument to the `run` call – this will completely switch off the "implicit namespace" mechanism.

The following three examples show the different ways of providing external variable values, all having the same effect in this case:

```
# Explicit argument to the NeuronGroup
G = NeuronGroup(1, 'dv/dt = -v / tau : 1', namespace={'tau': 10*ms})
net = Network(G)
net.run(10*ms)

# Explicit argument to the run function
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
net.run(10*ms, namespace={'tau': 10*ms})

# Implicit namespace from the context
```

(continues on next page)

```
G = NeuronGroup(1, 'dv/dt = -v / tau : 1')
net = Network(G)
tau = 10*ms
net.run(10*ms)
```

External variables are free to change between runs (but not during one run), the value at the time of the `run()` call is used in the simulation.

4.5 Custom progress reporting

4.5.1 Progress reporting

For custom progress reporting (e.g. graphical output, writing to a file, etc.), the `report` keyword accepts a callable (i.e. a function or an object with a `__call__` method) that will be called with four parameters:

- `elapsed`: the total (real) time since the start of the run

- `completed`: the fraction of the total simulation that is completed, i.e. a value between 0 and 1

- `start`: The start of the simulation (in biological time)

- `duration`: the total duration (in biological time) of the simulation

The function will be called every `report_period` during the simulation, but also at the beginning and end with `completed` equal to 0.0 and 1.0, respectively.

For the C++ standalone mode, the same standard options are available. It is also possible to implement custom progress reporting by directly passing the code (as a multi-line string) to the `report` argument. This code will be filled into a progress report function template, it should therefore only contain a function body. The simplest use of this might look like:

```
net.run(duration, report='std::cout << (int)(completed*100.) << "% completed" <<␣
→std::endl;')
```

Examples of custom reporting

Progress printed to a file

```
from brian2.core.network import TextReport
report_file = open('report.txt', 'w')
file_reporter = TextReport(report_file)
net.run(duration, report=file_reporter)
report_file.close()
```

"Graphical" output on the console

This needs a "normal" Linux console, i.e. it might not work in an integrated console in an IDE.

Adapted from http://stackoverflow.com/questions/3160699/python-progress-bar

```
import sys

class ProgressBar(object):
    def __init__(self, toolbar_width=40):
```

```python
        self.toolbar_width = toolbar_width
        self.ticks = 0

    def __call__(self, elapsed, complete, start, duration):
        if complete == 0.0:
            # setup toolbar
            sys.stdout.write("[%s]" % (" " * self.toolbar_width))
            sys.stdout.flush()
            sys.stdout.write("\b" * (self.toolbar_width + 1)) # return to start of
→line, after '['
        else:
            ticks_needed = int(round(complete * self.toolbar_width))
            if self.ticks < ticks_needed:
                sys.stdout.write("-" * (ticks_needed-self.ticks))
                sys.stdout.flush()
                self.ticks = ticks_needed
        if complete == 1.0:
            sys.stdout.write("\n")

net.run(duration, report=ProgressBar(), report_period=1*second)
```

"Standalone Mode" Text based progress bar on console

This needs a "normal" Linux console, i.e. it might not work in an integrated console in an IDE.

Adapted from https://stackoverflow.com/questions/14539867/how-to-display-a-progress-indicator-in-pure-c-c-cout-printf

```python
set_device('cpp_standalone')

report_func = '''
    int remaining = (int)((1-completed)/completed*elapsed+0.5);
    if (completed == 0.0)
    {
        std::cout << "Starting simulation at t=" << start << " s for duration " <<
→duration << " s"<<std::flush;
    }
    else
    {
        int barWidth = 70;
        std::cout << "\\r[";
        int pos = barWidth * completed;
        for (int i = 0; i < barWidth; ++i) {
                if (i < pos) std::cout << "=";
                else if (i == pos) std::cout << ">";
                else std::cout << " ";
        }
        std::cout << "] " << int(completed * 100.0) << "% completed. | "<
→<int(remaining) <<"s remaining"<<std::flush;
    }
'''
run(100*second, report=report_func)
```

4.6 Random numbers

Brian provides two basic functions to generate random numbers that can be used in model code and equations: `rand()`, to generate uniformly generated random numbers between 0 and 1, and `randn()`, to generate random numbers from a standard normal distribution (i.e. normally distributed numbers with a mean of 0 and a standard deviation of 1). All other stochastic elements of a simulation (probabilistic connections, Poisson-distributed input generated by *PoissonGroup* or *PoissonInput*, differential equations using the noise term `xi`, …) will internally make use of these two basic functions.

For *Runtime code generation*, random numbers are generated by `numpy.random.rand` and `numpy.random.randn` respectively, which uses a Mersenne-Twister pseudorandom number generator. When the `numpy` code generation target is used, these functions are called directly, but for `cython`, Brian uses a internal buffers for uniformly and normally distributed random numbers and calls the numpy functions whenever all numbers from this buffer have been used. This avoids the overhead of switching between C code and Python code for each random number. For *Standalone code generation*, the random number generation is based on "randomkit", the same Mersenne-Twister implementation that is used by numpy. The source code of this implementation will be included in every generated standalone project.

4.6.1 Seeding and reproducibility

Runtime mode

As explained above, *Runtime code generation* makes use of numpy's random number generator. In principle, using `numpy.random.seed` therefore permits reproducing a stream of random numbers. However, for `cython`, Brian's buffer complicates the matter a bit: if a simulation sets numpy's seed, uses 10000 random numbers, and then resets the seed, the following 10000 random numbers (assuming the current size of the buffer) will come out of the pre-generated buffer before numpy's random number generation functions are called again and take into account the seed set by the user. Instead, users should use the *seed()* function provided by Brian 2 itself, this will take care of setting numpy's random seed *and* empty Brian's internal buffers. This function also has the advantage that it will continue to work when the simulation is switched to standalone code generation (see below). Note that random numbers are not guaranteed to be reproducible across different code generation targets or different versions of Brian, especially if several sources of randomness are used in the same `CodeObject` (e.g. two noise variables in the equations of a *NeuronGroup*). This is because Brian does not guarantee the order of certain operations (e.g. should it first generate all random numbers for the first noise variable for all neurons, followed by the random numbers for the second noise variable for all neurons or rather first the random numbers for all noice variables of the first neuron, then for the second neuron, etc.) Since all random numbers are coming from the same stream of random numbers, the order of getting the numbers out of this stream matter.

Standalone mode

For *Standalone code generation*, Brian's *seed()* function will insert code to set the random number generator seed into the generated code. The code will be generated at the position where the *seed()* call was made, allowing detailed control over the seeding. For example the following code would generate identical initial conditions every time it is run, but the noise generated by the `xi` variable would differ:

```
G = NeuronGroup(10, 'dv/dt = -v/(10*ms) + 0.1*xi/sqrt(ms) : 1')
seed(4321)
G.v = 'rand()'
seed()
run(100*ms)
```

Note: In standalone mode, *seed()* will not set numpy's random number generator. If you use random numbers in the Python script itself (e.g. to generate a list of synaptic connections that will be passed to the standalone code as a

pre-calculated array), then you have to explicitly call `numpy.random.seed` yourself to make these random numbers reproducible.

Note: Seeding *should* lead to reproducible random numbers even when using OpenMP with multiple threads (for repeated simulations with the same number of threads), but this has not been rigorously tested. Use at your own risk.

4.7 Custom events

4.7.1 Overview

In most simulations, a *NeuronGroup* defines a threshold on its membrane potential that triggers a spike event. This event can be monitored by a *SpikeMonitor*, it is used in synaptic interactions, and in integrate-and-fire models it also leads to the execution of one or more reset statements.

Sometimes, it can be useful to define additional events, e.g. when an ion concentration in the cell crosses a certain threshold. This can be done with the custom events system in Brian, which is illustrated in this diagram.
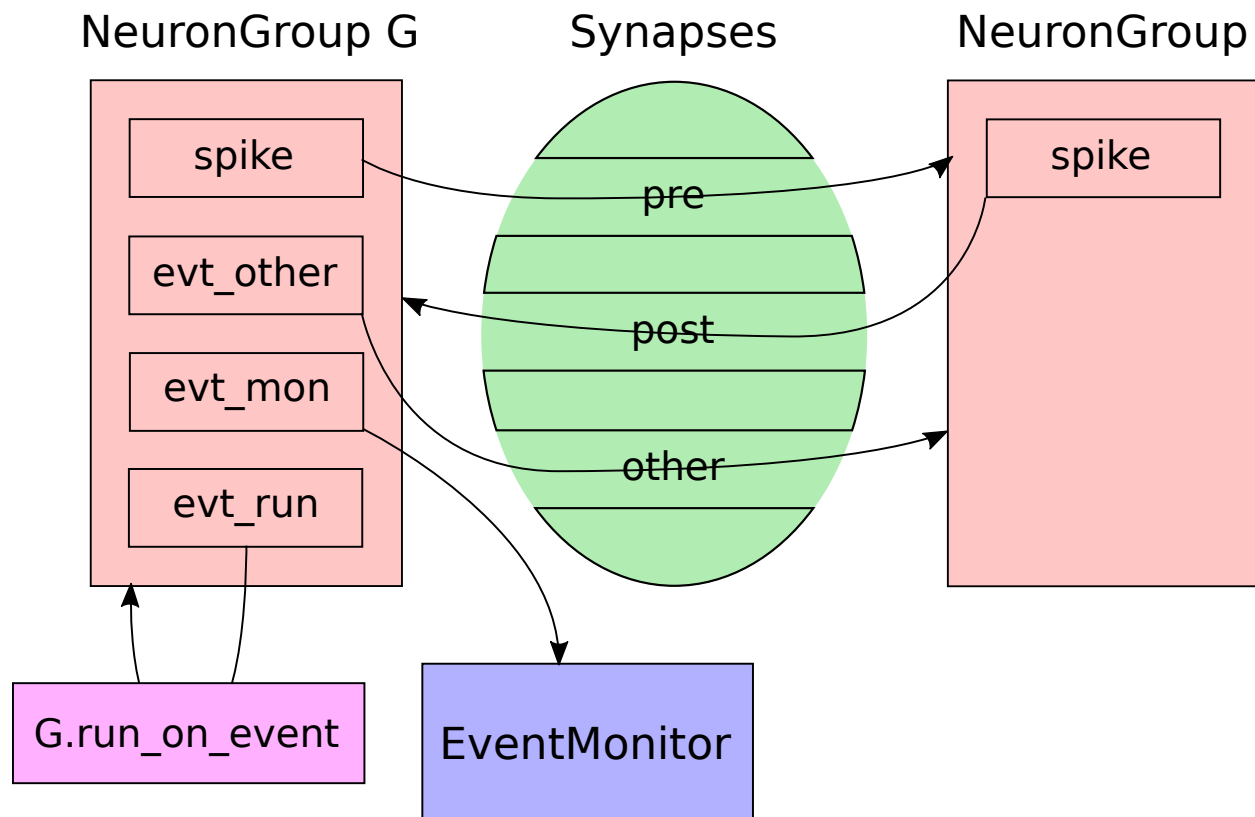
You can see in this diagram that the source *NeuronGroup* has four types of events, called `spike`, `evt_other`, `evt_mon` and `evt_run`. The event `spike` is the default event. It is triggered when you you include `threshold='.
..'` in a *NeuronGroup*, and has two potential effects. Firstly, when the event is triggered it causes the reset code to run, specified by `reset='...'`. Secondly, if there are *Synapses* connected, it causes the `on_pre` on `on_post` code to run (depending if the *NeuronGroup* is presynaptic or postsynaptic for those *Synapses*).

In the diagram though, we have three additional event types. We've included several event types here to make it clearer, but you could use the same event for different purposes. Let's start with the first one, `evt_other`. To understand this,

---

we need to look at the *Synapses* object in a bit more detail. A *Synapses* object has multiple *pathways* associated to it. By default, there are just two, called `pre` and `post`. The `pre` pathway is activated by presynaptic spikes, and the `post` pathway by postsynaptic spikes. Specifically, the `spike` event on the presynaptic *NeuronGroup* triggers the `pre` pathway, and the `spike` event on the postsynaptic *NeuronGroup* triggers the `post` pathway. In the example in the diagram, we have created a new pathway called `other`, and the `evt_other` event in the presynaptic *NeuronGroup* triggers this pathway. Note that we can arrange this however we want. We could have `spike` trigger the `other` pathway if we wanted to, or allow it to trigger both the `pre` and `other` pathways. We could also allow `evt_other` to trigger the `pre` pathway. See below for details on the syntax for this.

The third type of event in the example is named `evt_mon` and this is connected to an *EventMonitor* which works exactly the same way as *SpikeMonitor* (which is just an *EventMonitor* attached by default to the event `spike`).

Finally, the fourth type of event in the example is named `evt_run`, and this causes some code to be run in the *NeuronGroup* triggered by the event. To add this code, we call *NeuronGroup.run_on_event*. So, when you set `reset='...'`, this is equivalent to calling *NeuronGroup.run_on_event* with the `spike` event.

4.7.2 Details

Defining an event

This can be done with the `events` keyword in the *NeuronGroup* initializer:

```
group = NeuronGroup(N, '...', threshold='...', reset='...',
                    events={'custom_event': 'x > x_th'})
```

In this example, we define an event with the name `custom_event` that is triggered when the `x` variable crosses the threshold `x_th`. Note that you can define any number of custom events. Each event is defined by its name as the key, and its condition as the value of the dictionary.

Recording events

Custom events can be recorded with an *EventMonitor*:

```
event_mon = EventMonitor(group, 'custom_event')
```

Such an *EventMonitor* can be used in the same way as a *SpikeMonitor* – in fact, creating the *SpikeMonitor* is basically identical to recording the `spike` event with an *EventMonitor*. An *EventMonitor* is not limited to record the event time/neuron index, it can also record other variables of the model at the time of the event:

```
event_mon = EventMonitor(group, 'custom_event', variables['var1', 'var2'])
```

Triggering `NeuronGroup` code

If the event should trigger a series of statements (i.e. the equivalent of `reset` statements), this can be added by calling `run_on_event`:

```
group.run_on_event('custom_event', 'x=0')
```

Triggering synaptic pathways

When neurons are connected by *Synapses*, the `pre` and `post` pathways are triggered by `spike` events on the presynaptic and postsynaptic *NeuronGroup* by default. It is possible to change which pathway is triggered by which event by providing an `on_event` keyword that either specifies which event to use for all pathways, or a specific event for each pathway (where non-specified pathways use the default `spike` event):

```
synapse_1 = Synapses(group, another_group, '...', on_pre='...', on_event='custom_event
→')
```

The code above causes all pathways to be triggered by an event named `custom_event` instead of the default `spike`.

```
synapse_2 = Synapses(group, another_group, '...', on_pre='...', on_post='...',
                     on_event={'pre': 'custom_event'})
```

In the code above, only the `pre` pathway is triggered by the `custom_event` event.

We can also create new pathways and have them be triggered by custom events. For example:

```
synapse_3 = Synapses(group, another_group, '...',
                     on_pre={'pre': '....',
                             'custom_pathway': '...'},
                     on_event={'pre': 'spike',
                               'custom_pathway': 'custom_event'})
```

In this code, the default `pre` pathway is still triggered by the `spike` event, but there is a new pathway called `custom_pathway` that is triggered by the `custom_event` event.

Scheduling

By default, custom events are checked after the spiking threshold (in the `after_thresholds` slots) and statements are executed after the reset (in the `after_resets` slots). The slot for the execution of custom event-triggered statements can be changed when it is added with the usual `when` and `order` keyword arguments (see *Scheduling* for details). To change the time when the condition is checked, use *NeuronGroup.set_event_schedule*.

4.8 State update

In Brian, a state updater transforms a set of equations into an abstract state update code (and therefore is automatically target-independent). In general, any function (or callable object) that takes an *Equations* object and returns abstract code (as a string) can be used as a state updater and passed to the *NeuronGroup* constructor as a `method` argument.

The more common use case is to specify no state updater at all or chose one by name, see *Choice of state updaters* below.

4.8.1 Explicit state update

Explicit state update schemes can be specified in mathematical notation, using the *ExplicitStateUpdater* class. A state updater scheme contains a series of statements, defining temporary variables and a final line (starting with `x_new` =), giving the updated value for the state variable. The description can make reference to `t` (the current time), `dt` (the size of the time step), `x` (value of the state variable), and `f(x, t)` (the definition of the state variable `x`, assuming $dx/dt = f(x, t)$. In addition, state updaters supporting stochastic equations additionally make use of `dW` (a normal distributed random variable with variance `dt`) and `g(x, t)`, the factor multiplied with the noise variable, assuming $dx/dt = f(x, t) + g(x, t) * xi$.

Using this notation, simple forward Euler integration is specified as:

```
x_new = x + dt * f(x, t)
```

A Runge-Kutta 2 (midpoint) method is specified as:

```
k = dt * f(x,t)
x_new = x + dt * f(x +  k/2, t + dt/2)
```

When creating a new state updater using *ExplicitStateUpdater*, you can specify the `stochastic` keyword argument, determining whether this state updater does not support any stochastic equations (`None`, the default), stochastic equations with additive noise only (`'additive'`), or arbitrary stochastic equations (`'multiplicative'`). The provided state updaters use the Stratonovich interpretation for stochastic equations (which is the correct interpretation if the white noise source is seen as the limit of a coloured noise source with a short time constant). As a result of this, the simple Euler-Maruyama scheme (`x_new = x + dt*f(x, t) + dW*g(x, t)`) will only be used for additive noise.

An example for a general state updater that handles arbitrary multiplicative noise (under Stratonovich interpretation) is the derivative-free Milstein method:

```
x_support = x + dt*f(x, t) + dt**.5 * g(x, t)
g_support = g(x_support, t)
k = 1/(2*dt**.5)*(g_support - g(x, t))*(dW**2)
x_new = x + dt*f(x,t) + g(x, t) * dW + k
```

Note that a single line in these descriptions is only allowed to mention `g(x, t)`, respectively `f(x, t)` only once (and you are not allowed to write, for example, `g(f(x, t), t)`). You can work around these restrictions by using intermediate steps, defining temporary variables, as in the above examples for *milstein* and *rk2*.

4.8.2 Choice of state updaters

As mentioned in the beginning, you can pass arbitrary callables to the method argument of a *NeuronGroup*, as long as this callable converts an *Equations* object into abstract code. The best way to add a new state updater, however, is to register it with brian and provide a method to determine whether it is appropriate for a given set of equations. This way, it can be automatically chosen when no method is specified and it can be referred to with a name (i.e. you can pass a string like `'euler'` to the method argument instead of importing *euler* and passing a reference to the object itself).

If you create a new state updater using the *ExplicitStateUpdater* class, you have to specify what kind of stochastic equations it supports. The keyword argument `stochastic` takes the values `None` (no stochastic equation support, the default), `'additive'` (support for stochastic equations with additive noise), `'multiplicative'` (support for arbitrary stochastic equations).

After creating the state updater, it has to be registered with *StateUpdateMethod*:

```
new_state_updater = ExplicitStateUpdater('...', stochastic='additive')
StateUpdateMethod.register('mymethod', new_state_updater)
```

The preferred way to do write new general state updaters (i.e. state updaters that cannot be described using the explicit syntax described above) is to extend the *StateUpdateMethod* class (but this is not strictly necessary, all that is needed is an object that implements a `__call__` method that operates on an *Equations* object and a dictionary of variables). Optionally, the state updater can be registered with *StateUpdateMethod* as shown above.

4.8.3 Implicit state updates

Note: All of the following is just here for future reference, it's not implemented yet.

Implicit schemes often use Newton-Raphson or fixed point iterations. These can also be defined by mathematical statements, but the number of iterations is dynamic and therefore not easily vectorised. However, this might not be a big issue in C, GPU or even with Numba.

Backward Euler

Backward Euler is defined as follows:

```
x(t+dt)=x(t)+dt*f(x(t+dt),t+dt)
```

This is not a executable statement because the RHS depends on the future. A simple way is to perform fixed point iterations:

```
x(t+dt)=x(t)
x(t+dt)=x(t)+dt*dx=f(x(t+dt),t+dt)    until increment<tolerance
```

This includes a loop with a different number of iterations depending on the neuron.

4.9 How Brian works

In this section we will briefly cover some of the internals of how Brian works. This is included here to understand the general process that Brian goes through in running a simulation, but it will not be sufficient to understand the source code of Brian itself or to extend it to do new things. For a more detailed view of this, see the documentation in the *Developer's guide*.

4.9.1 Clock-driven versus event-driven

Brian is a clock-driven simulator. This means that the simulation time is broken into an equally spaced time grid, 0, dt, 2*dt, 3*dt, …. At each time step t, the differential equations specifying the models are first integrated giving the values at time t+dt. Spikes are generated when a condition such as $v>vt$ is satisfied, and spikes can only occur on the time grid.

The advantage of clock driven simulation is that it is very flexible (arbitrary differential equations can be used) and computationally efficient. However, the time grid approximation can lead to an overestimate of the amount of synchrony that is present in a network. This is usually not a problem, and can be managed by reducing the time step dt, but it can be an issue for some models.

Note that the inaccuracy introduced by the spike time approximation is of order O(dt), so the total accuracy of the simulation is of order O(dt) per time step. This means that in many cases, there is no need to use a higher order numerical integration method than forward Euler, as it will not improve the order of the error beyond O(dt). See *State update* for more details of numerical integration methods.

Some simulators use an event-driven method. With this method, spikes can occur at arbitrary times instead of just on the grid. This method can be more accurate than a clock-driven simulation, but it is usually substantially more computationally expensive (especially for larger networks). In addition, they are usually more restrictive in terms of the class of differential equations that can be solved.

For a review of some of the simulation strategies that have been used, see Brette et al. 2007.

4.9.2 Code overview

The user-visible part of Brian consists of a number of objects such as *NeuronGroup*, *Synapses*, *Network*, etc. These are all written in pure Python and essentially work to translate the user specified model into the computational engine. The end state of this translation is a collection of short blocks of code operating on a namespace, which are called in a sequence by the *Network*. Examples of these short blocks of code are the "state updaters" which perform numerical integration, or the synaptic propagation step. The namespaces consist of a mapping from names to values, where the possible values can be scalar values, fixed-length or dynamically sized arrays, and functions.

4.9.3 Syntax layer

The syntax layer consists of everything that is independent of the way the final simulation is computed (i.e. the language and device it is running on). This includes things like *NeuronGroup*, *Synapses*, *Network*, *Equations*, etc.

The user-visible part of this is documented fully in the *User's guide* and the *Advanced guide*. In particular, things such as the analysis of equations and assignment of numerical integrators. The end result of this process, which is passed to the computational engine, is a specification of the simulation consisting of the following data:

- A collection of variables which are scalar values, fixed-length arrays, dynamically sized arrays, and functions. These are handled by `Variable` objects detailed in *Variables and indices*. Examples: each state variable of a *NeuronGroup* is assigned an `ArrayVariable`; the list of spike indices stored by a *SpikeMonitor* is assigned a `DynamicArrayVariable`; etc.

- A collection of code blocks specified via an "abstract code block" and a template name. The "abstract code block" is a sequence of statements such as `v = vr` which are to be executed. In the case that say, `v` and `vr` are arrays, then the statement is to be executed for each element of the array. These abstract code blocks are either given directly by the user (in the case of neuron threshold and reset, and synaptic pre and post codes), or generated from differential equations combined with a numerical integrator. The template name is one of a small set (around 20 total) which give additional context. For example, the code block `a = b` when considered as part of a "state update" means execute that for each neuron index. In the context of a reset statement, it means execute it for each neuron index of a neuron that has spiked. Internally, these templates need to be implemented for each target language/device, but there are relatively few of them.

- The order of execution of these code blocks, as defined by the *Network*.

4.9.4 Computational engine

The computational engine covers everything from generating to running code in a particular language or on a particular device. It starts with the abstract definition of the simulation resulting from the syntax layer described above.

The computational engine is described by a `Device` object. This is used for allocating memory, generating and running code. There are two types of device, "runtime" and "standalone". In runtime mode, everything is managed by Python, even if individual code blocks are in a different language. Memory is managed using numpy arrays (which can be passed as pointers to use in other languages). In standalone mode, the output of the process (after calling `Device.build`) is a complete source code project that handles everything, including memory management, and is independent of Python.

For both types of device, one of the key steps that works in the same way is code generation, the creation of a compilable and runnable block of code from an abstract code block and a collection of variables. This happens in two stages: first of all, the abstract code block is converted into a code snippet, which is a syntactically correct block of code in the target language, but not one that can run on its own (it doesn't handle accessing the variables from memory, etc.). This code snippet typically represents the inner loop code. This step is handled by a `CodeGenerator` object. In some cases it will involve a syntax translation (e.g. the Python syntax `x**y` in C++ should be `pow(x, y)`). The next step is to insert this code snippet into a template to form a compilable code block. This code block is then passed to a runtime `CodeObject`. In the case of standalone mode, this doesn't do anything, but for runtime devices it handles compiling the code and then running the compiled code block in the given namespace.

4.10 Interfacing with external code

Some neural simulations benefit from a direct connections to external libraries, e.g. to support real-time input from a sensor (but note that Brian currently does not offer facilities to assure real-time processing) or to perform complex calculations during a simulation run.

If the external library is written in Python (or is a library with Python bindings), then the connection can be made either using the mechanism for *User-provided functions*, or using a *network operation*.

In case of C/C++ libraries, only the *User-provided functions* mechanism can be used. On the other hand, such simulations can use the same user-provided C++ code to run with the *Standalone code generation* mode. In addition to that code, one generally needs to include additional header files and use compiler/linker options to interface with the external code. For this, several preferences can be used that will be taken into account for `cython` and the `cpp_standalone` device. These preferences are mostly equivalent to the respective keyword arguments for Python's `distutils.core.Extension` class, see the documentation of the `cpp_prefs` module for more details.

EXAMPLES

5.1 Example: COBAHH

This is an implementation of a benchmark described in the following review paper:

Simulation of networks of spiking neurons: A review of tools and strategies (2006). Brette, Rudolph, Carnevale, Hines, Beeman, Bower, Diesmann, Goodman, Harris, Zirpe, Natschläger, Pecevski, Ermentrout, Djurfeldt, Lansner, Rochel, Vibert, Alvarez, Muller, Davison, El Boustani and Destexhe. Journal of Computational Neuroscience

Benchmark 3: random network of HH neurons with exponential synaptic conductances

Clock-driven implementation (no spike time interpolation)

R. Brette - Dec 2007

```python
from brian2 import *

# Parameters
area = 20000*umetre**2
Cm = (1*ufarad*cm**-2) * area
gl = (5e-5*siemens*cm**-2) * area

El = -60*mV
EK = -90*mV
ENa = 50*mV
g_na = (100*msiemens*cm**-2) * area
g_kd = (30*msiemens*cm**-2) * area
VT = -63*mV
# Time constants
taue = 5*ms
taui = 10*ms
# Reversal potentials
Ee = 0*mV
Ei = -80*mV
we = 6*nS  # excitatory synaptic weight
wi = 67*nS  # inhibitory synaptic weight

# The model
eqs = Equations('''
dv/dt = (gl*(El-v)+ge*(Ee-v)+gi*(Ei-v)-
         g_na*(m*m*m)*h*(v-ENa)-
         g_kd*(n*n*n*n)*(v-EK))/Cm : volt
dm/dt = alpha_m*(1-m)-beta_m*m : 1
```

(continues on next page)

```
dn/dt = alpha_n*(1-n)-beta_n*n : 1
dh/dt = alpha_h*(1-h)-beta_h*h : 1
dge/dt = -ge*(1./taue) : siemens
dgi/dt = -gi*(1./taui) : siemens
alpha_m = 0.32*(mV**-1)*4*mV/exprel((13*mV-v+VT)/(4*mV))/ms : Hz
beta_m = 0.28*(mV**-1)*5*mV/exprel((v-VT-40*mV)/(5*mV))/ms : Hz
alpha_h = 0.128*exp((17*mV-v+VT)/(18*mV))/ms : Hz
beta_h = 4./(1+exp((40*mV-v+VT)/(5*mV)))/ms : Hz
alpha_n = 0.032*(mV**-1)*5*mV/exprel((15*mV-v+VT)/(5*mV))/ms : Hz
beta_n = .5*exp((10*mV-v+VT)/(40*mV))/ms : Hz
''')

P = NeuronGroup(4000, model=eqs, threshold='v>-20*mV', refractory=3*ms,
                method='exponential_euler')
Pe = P[:3200]
Pi = P[3200:]
Ce = Synapses(Pe, P, on_pre='ge+=we')
Ci = Synapses(Pi, P, on_pre='gi+=wi')
Ce.connect(p=0.02)
Ci.connect(p=0.02)

# Initialization
P.v = 'El + (randn() * 5 - 5)*mV'
P.ge = '(randn() * 1.5 + 4) * 10.*nS'
P.gi = '(randn() * 12 + 20) * 10.*nS'

# Record a few traces
trace = StateMonitor(P, 'v', record=[1, 10, 100])
run(1 * second, report='text')
plot(trace.t/ms, trace[1].v/mV)
plot(trace.t/ms, trace[10].v/mV)
plot(trace.t/ms, trace[100].v/mV)
xlabel('t (ms)')
ylabel('v (mV)')
show()
```
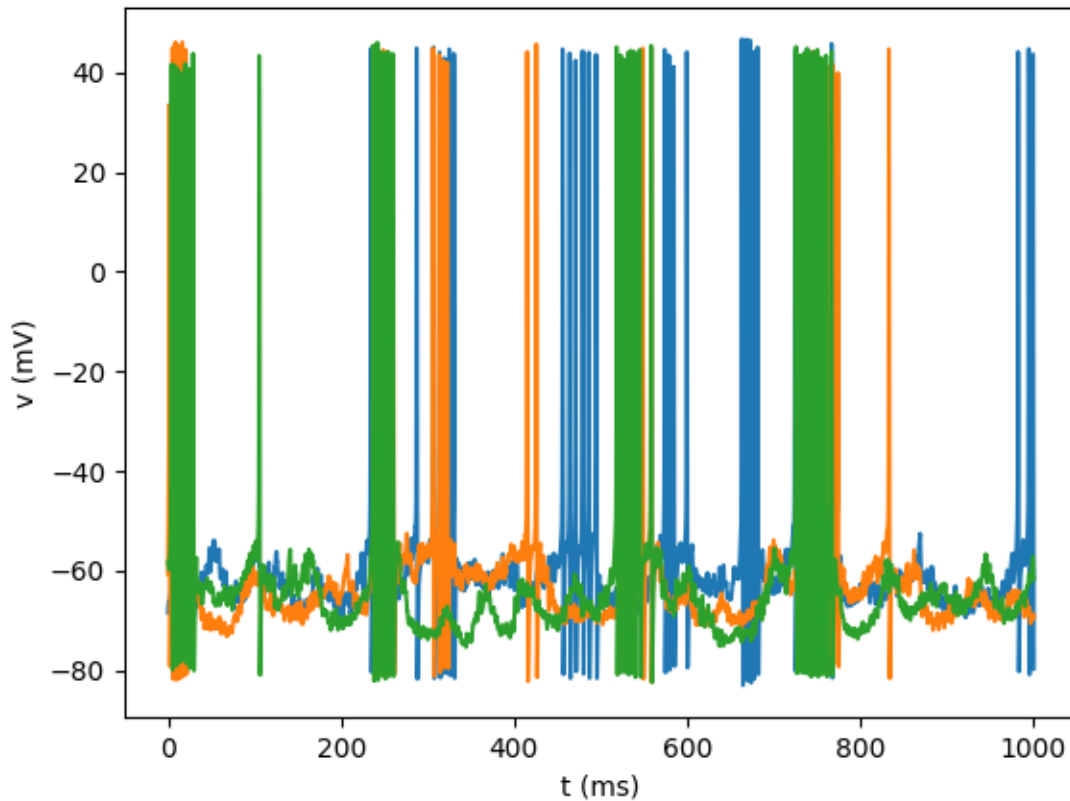
5.2 Example: CUBA

This is a Brian script implementing a benchmark described in the following review paper:

Simulation of networks of spiking neurons: A review of tools and strategies (2007). Brette, Rudolph, Carnevale, Hines, Beeman, Bower, Diesmann, Goodman, Harris, Zirpe, Natschlager, Pecevski, Ermentrout, Djurfeldt, Lansner, Rochel, Vibert, Alvarez, Muller, Davison, El Boustani and Destexhe. Journal of Computational Neuroscience 23(3):349-98

Benchmark 2: random network of integrate-and-fire neurons with exponential synaptic currents.

Clock-driven implementation with exact subthreshold integration (but spike times are aligned to the grid).

```python
from brian2 import *

taum = 20*ms
taue = 5*ms
taui = 10*ms
Vt = -50*mV
Vr = -60*mV
El = -49*mV

eqs = '''
```

5.3 Example: IF_curve_Hodgkin_Huxley

Input-Frequency curve of a HH model.

Network: 100 unconnected Hodgin-Huxley neurons with an input current I. The input is set differently for each neuron.

This simulation should use exponential Euler integration.

```python
from brian2 import *

num_neurons = 100
duration = 2*second

# Parameters
area = 20000*umetre**2
Cm = 1*ufarad*cm**-2 * area
gl = 5e-5*siemens*cm**-2 * area
El = -65*mV
EK = -90*mV
ENa = 50*mV
g_na = 100*msiemens*cm**-2 * area
g_kd = 30*msiemens*cm**-2 * area
```

(continues on next page)

```python
VT = -63*mV

# The model
eqs = Equations('''
dv/dt = (gl*(El-v) - g_na*(m*m*m)*h*(v-ENa) - g_kd*(n*n*n*n)*(v-EK) + I)/Cm : volt
dm/dt = 0.32*(mV**-1)*4*mV/exprel((13.*mV-v+VT)/(4*mV))/ms*(1-m)-0.28*(mV**-1)*5*mV/
→exprel((v-VT-40.*mV)/(5*mV))/ms*m : 1
dn/dt = 0.032*(mV**-1)*5*mV/exprel((15.*mV-v+VT)/(5*mV))/ms*(1.-n)-.5*exp((10.*mV-
→v+VT)/(40.*mV))/ms*n : 1
dh/dt = 0.128*exp((17.*mV-v+VT)/(18.*mV))/ms*(1.-h)-4./(1+exp((40.*mV-v+VT)/(5.*mV)))/
→ms*h : 1
I : amp
''')
# Threshold and refractoriness are only used for spike counting
group = NeuronGroup(num_neurons, eqs,
                    threshold='v > -40*mV',
                    refractory='v > -40*mV',
                    method='exponential_euler')
group.v = El
group.I = '0.7*nA * i / num_neurons'

monitor = SpikeMonitor(group)

run(duration)

plot(group.I/nA, monitor.count / duration)
xlabel('I (nA)')
ylabel('Firing rate (sp/s)')
show()
```
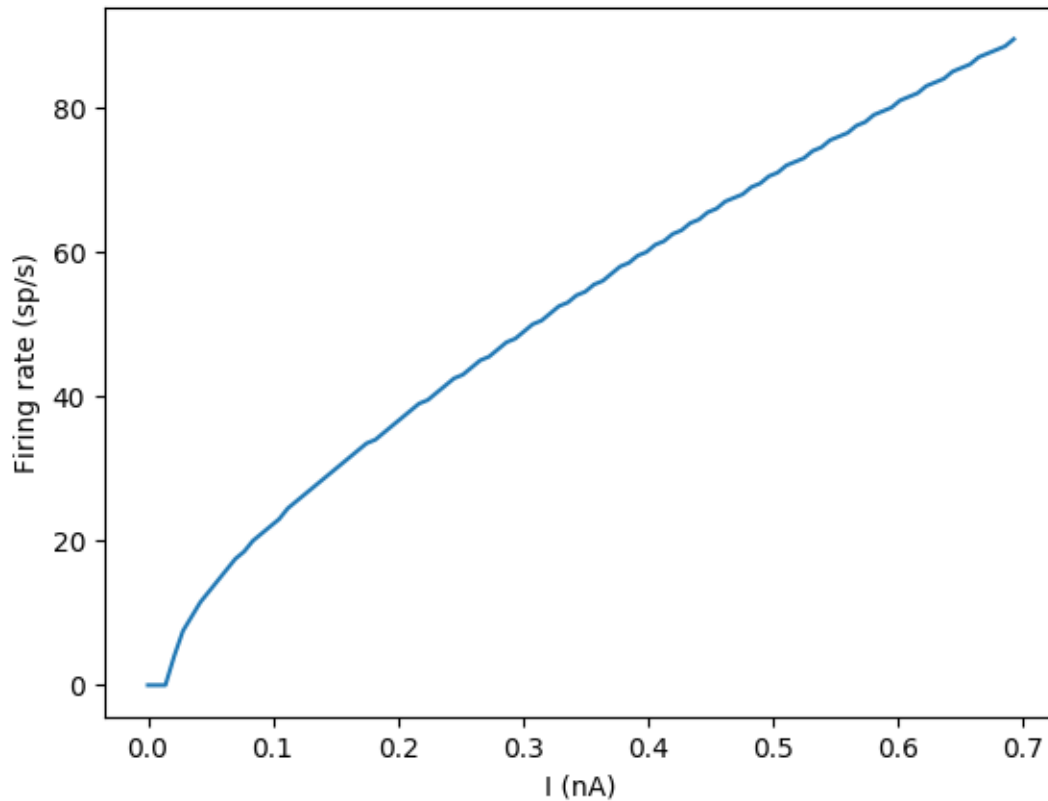
5.4 Example: IF_curve_LIF

Input-Frequency curve of a IF model.

Network: 1000 unconnected integrate-and-fire neurons (leaky IF) with an input parameter v0. The input is set differently for each neuron.

```python
from brian2 import *

n = 1000
duration = 1*second
tau = 10*ms
eqs = '''
dv/dt = (v0 - v) / tau : volt (unless refractory)
v0 : volt
'''
group = NeuronGroup(n, eqs, threshold='v > 10*mV', reset='v = 0*mV',
                    refractory=5*ms, method='exact')
group.v = 0*mV
group.v0 = '20*mV * i / (n-1)'

monitor = SpikeMonitor(group)
```

```
run(duration)
plot(group.v0/mV, monitor.count / duration)
xlabel('v0 (mV)')
ylabel('Firing rate (sp/s)')
show()
```
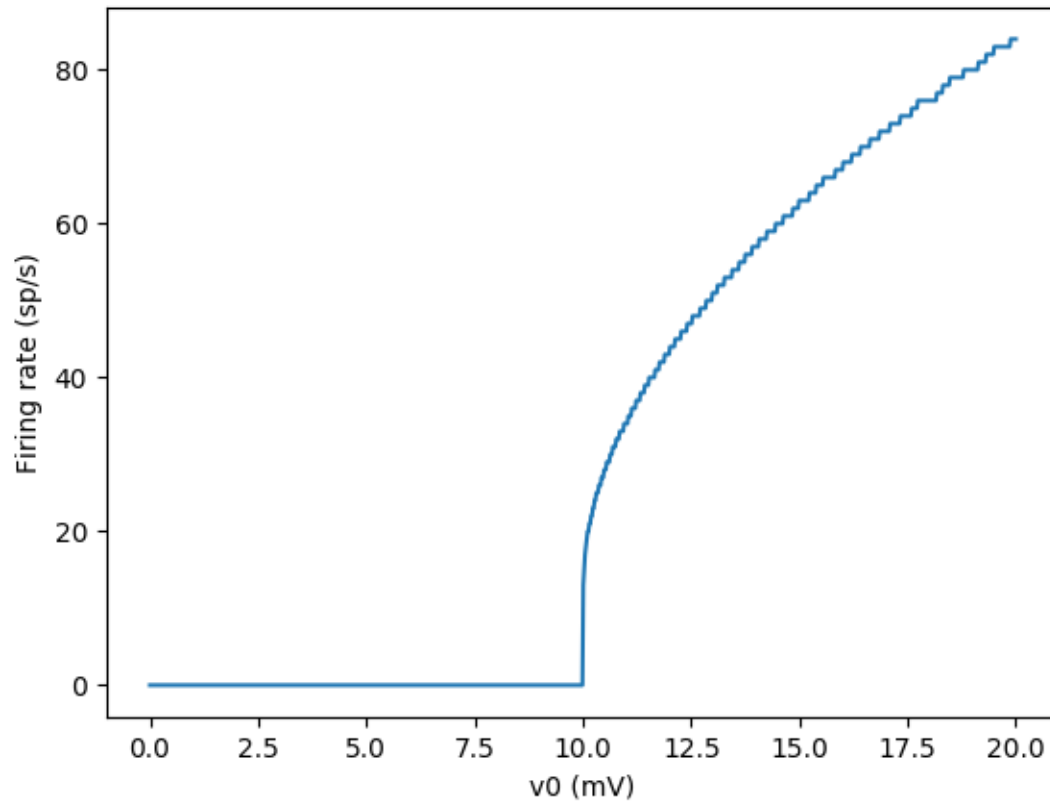

5.5 Example: adaptive_threshold

A model with adaptive threshold (increases with each spike)

```
from brian2 import *

eqs = '''
dv/dt = -v/(10*ms) : volt
dvt/dt = (10*mV-vt)/(15*ms) : volt
'''

reset = '''
v = 0*mV
```

```python
vt += 3*mV
'''

IF = NeuronGroup(1, model=eqs, reset=reset, threshold='v>vt',
                 method='exact')
IF.vt = 10*mV
PG = PoissonGroup(1, 500 * Hz)

C = Synapses(PG, IF, on_pre='v += 3*mV')
C.connect()

Mv = StateMonitor(IF, 'v', record=True)
Mvt = StateMonitor(IF, 'vt', record=True)
# Record the value of v when the threshold is crossed
M_crossings = SpikeMonitor(IF, variables='v')
run(2*second, report='text')

subplot(1, 2, 1)
plot(Mv.t / ms, Mv[0].v / mV)
plot(Mvt.t / ms, Mvt[0].vt / mV)
ylabel('v (mV)')
xlabel('t (ms)')
# zoom in on the first 100ms
xlim(0, 100)
subplot(1, 2, 2)
hist(M_crossings.v / mV, bins=np.arange(10, 20, 0.5))
xlabel('v at threshold crossing (mV)')
show()
```
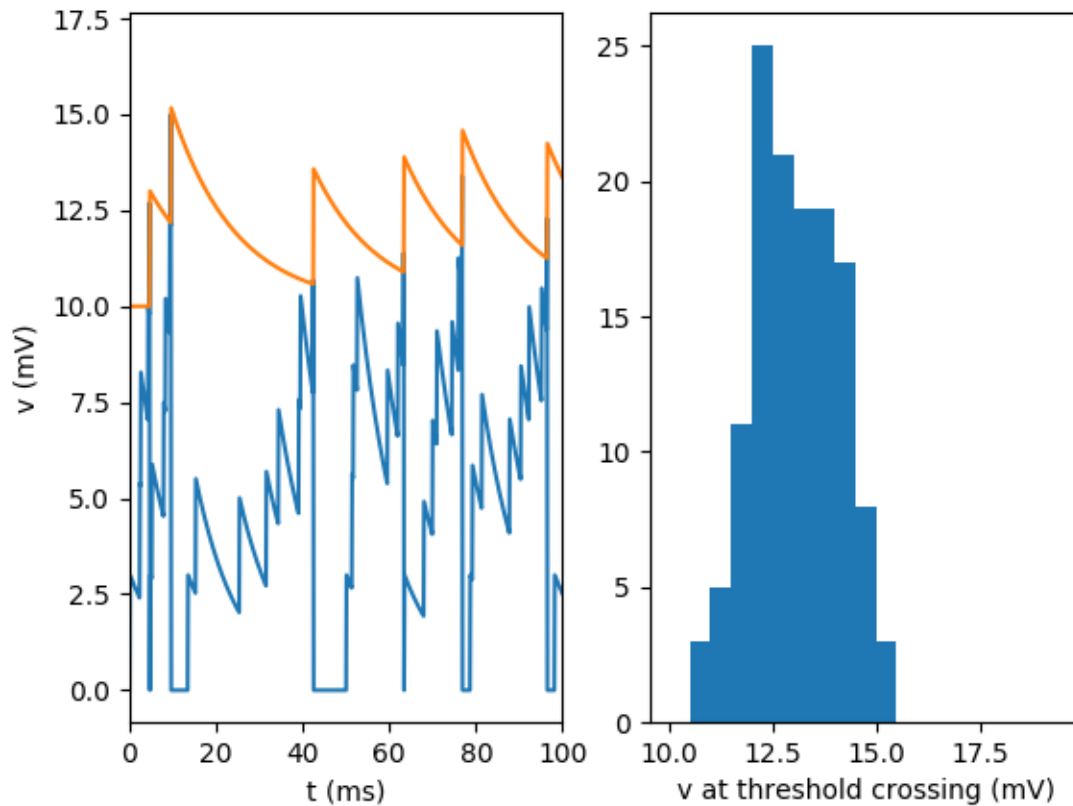
5.6 Example: non_reliability

Reliability of spike timing.

See e.g. Mainen & Sejnowski (1995) for experimental results in vitro.

Here: a constant current is injected in all trials.

```python
from brian2 import *

N = 25
tau = 20*ms
sigma = .015
eqs_neurons = '''
dx/dt = (1.1 - x) / tau + sigma * (2 / tau)**.5 * xi : 1 (unless refractory)
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1', reset='x = 0',
                      refractory=5*ms, method='euler')
spikes = SpikeMonitor(neurons)

run(500*ms)
plot(spikes.t/ms, spikes.i, '.k')
```

```
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```
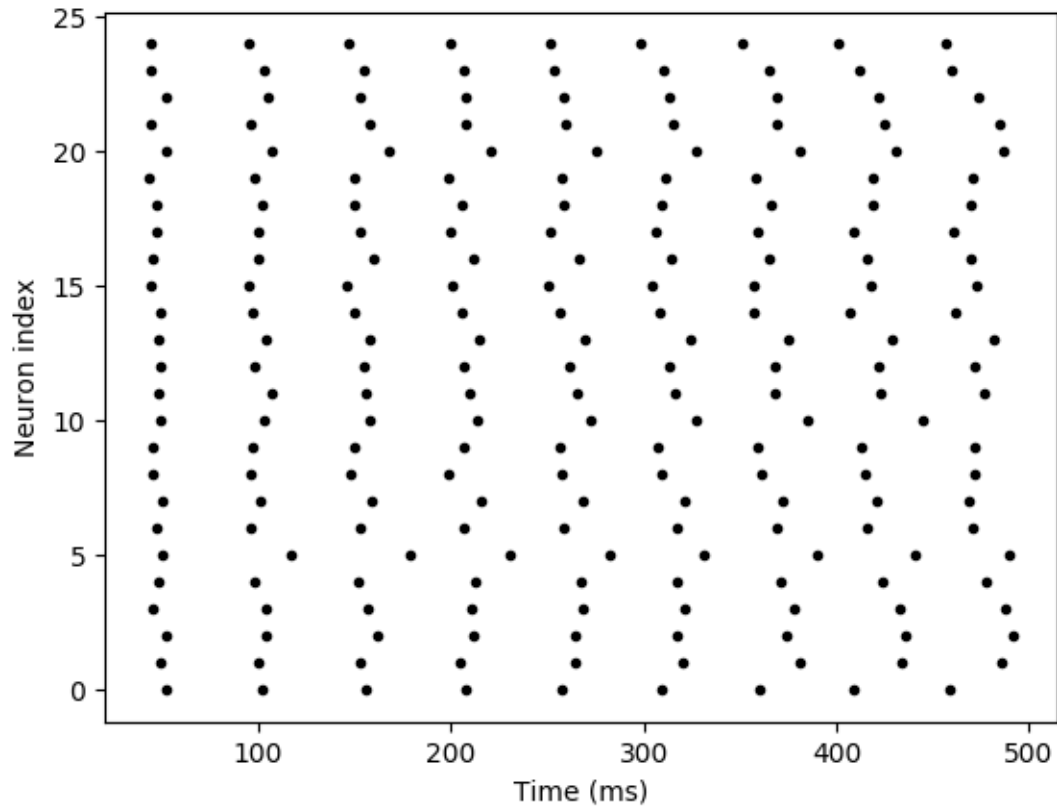

5.7 Example: phase_locking

Phase locking of IF neurons to a periodic input.

```python
from brian2 import *

tau = 20*ms
n = 100
b = 1.2 # constant current mean, the modulation varies
freq = 10*Hz

eqs = '''
dv/dt = (-v + a * sin(2 * pi * freq * t) + b) / tau : 1
a : 1
'''
neurons = NeuronGroup(n, model=eqs, threshold='v > 1', reset='v = 0',
```

```
                        method='euler')
neurons.v = 'rand()'
neurons.a = '0.05 + 0.7*i/n'
S = SpikeMonitor(neurons)
trace = StateMonitor(neurons, 'v', record=50)

run(1000*ms)
subplot(211)
plot(S.t/ms, S.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
subplot(212)
plot(trace.t/ms, trace.v.T)
xlabel('Time (ms)')
ylabel('v')
tight_layout()
show()
```
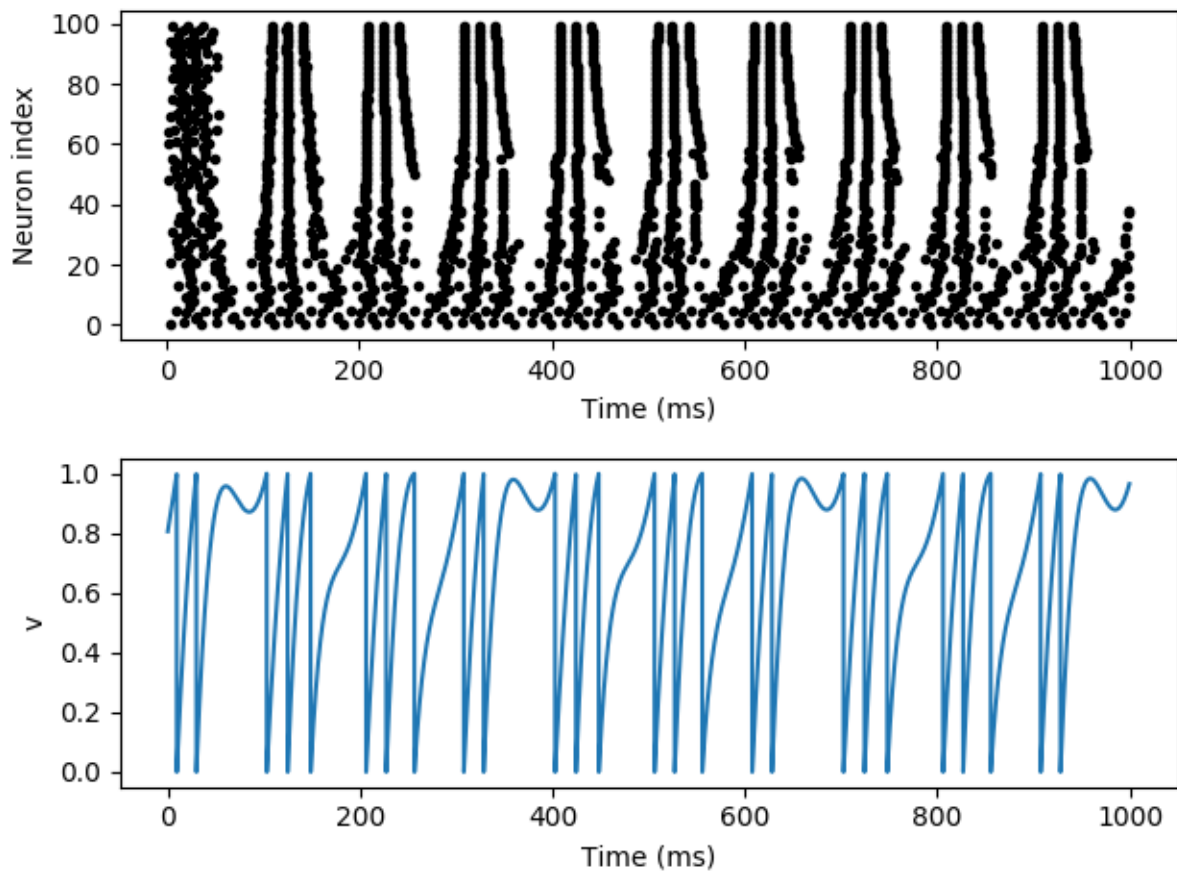
5.8 Example: reliability

Reliability of spike timing.

See e.g. Mainen & Sejnowski (1995) for experimental results in vitro.

```python
from brian2 import *

# The common noisy input
N = 25
tau_input = 5*ms
neuron_input = NeuronGroup(1, 'dx/dt = -x / tau_input + (2 /tau_input)**.5 * xi : 1')

# The noisy neurons receiving the same input
tau = 10*ms
sigma = .015
eqs_neurons = '''
dx/dt = (0.9 + .5 * I - x) / tau + sigma * (2 / tau)**.5 * xi : 1
I : 1 (linked)
'''
neurons = NeuronGroup(N, model=eqs_neurons, threshold='x > 1',
                      reset='x = 0', refractory=5*ms, method='euler')
neurons.x = 'rand()'
neurons.I = linked_var(neuron_input, 'x') # input.x is continuously fed into neurons.I
spikes = SpikeMonitor(neurons)

run(500*ms)
plt.plot(spikes.t/ms, spikes.i, '.k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```
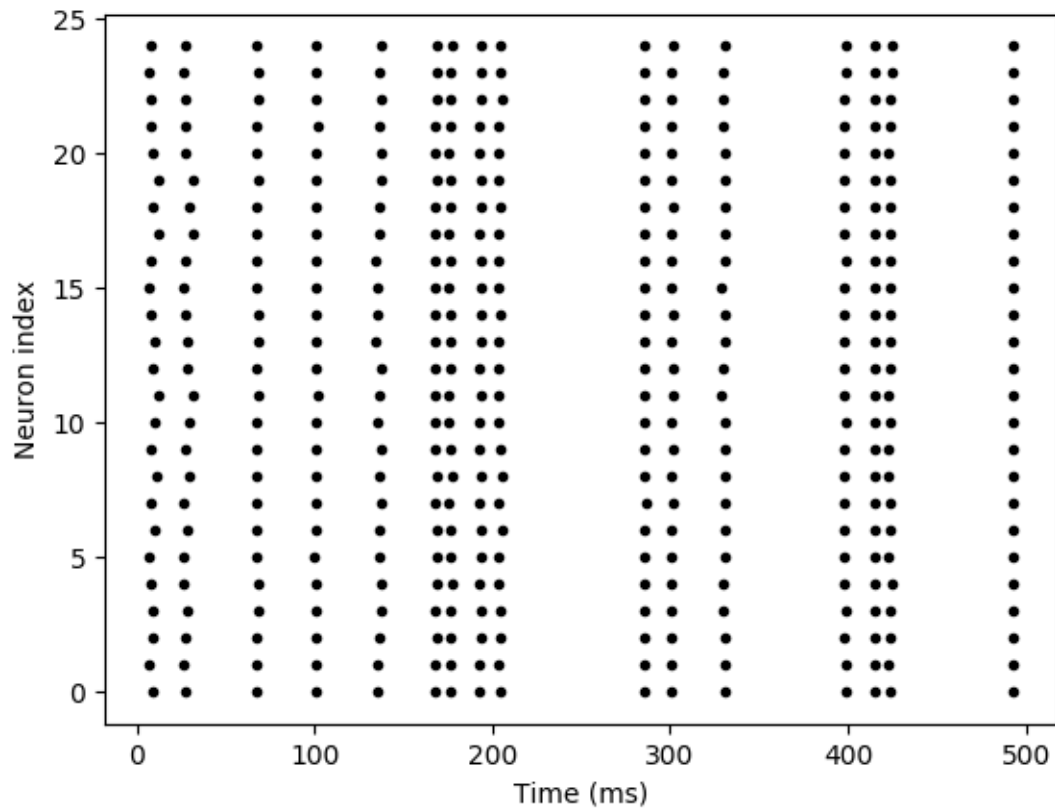
5.9 advanced

5.9.1 Example: COBAHH_approximated

Follows exercise 4, chapter 2 of Eugene M. Izhikevich: Dynamical Systems in Neuroscience

Sebastian Schmitt, 2021

```python
import argparse
from functools import reduce
import operator

import matplotlib.pyplot as plt
from cycler import cycler
import numpy as np

from brian2 import run
from brian2 import mS, cmeter, ms, mV, uA, uF
from brian2 import Equations, NeuronGroup, StateMonitor, TimedArray, defaultclock
```

```python
def construct_gating_variable_inf_equation(gating_variable):
    """Construct the voltage-dependent steady-state gating variable equation.

    Approximated by Boltzmann function.

    gating_variable -- gating variable, typically one of "m", "n" and "h"
    """

    return Equations('xinf = 1/(1+exp((v_half-v)/k)) : 1',
                     xinf=f'{gating_variable}_inf',
                     v_half=f'v_{gating_variable}_half',
                     k=f'k_{gating_variable}')


def construct_gating_variable_tau_equation(gating_variable):
    """Construct the voltage-dependent gating variable time constant equation.

    Approximated by Gaussian function.

    gating_variable -- gating variable, typically one of "m", "n" and "h"
    """

    return Equations('tau = c_base + c_amp*exp(-(v_max - v)**2/sigma**2) : second',
                     tau=f'tau_{gating_variable}',
                     c_base=f'c_{gating_variable}_base',
                     c_amp=f'c_{gating_variable}_amp',
                     v_max=f'v_{gating_variable}_max',
                     sigma=f'sigma_{gating_variable}')


def construct_gating_variable_ode(gating_variable):
    """Construct the ordinary differential equation of the gating variable.

    gating_variable -- gating variable, typically one of "m", "n" and "h"
    """

    return Equations('dx/dt = (xinf - x)/tau : 1',
                     x=gating_variable,
                     xinf=f'{gating_variable}_inf',
                     tau=f'tau_{gating_variable}')


def construct_neuron_ode():
    """Construct the ordinary differential equation of the membrane."""

    # conductances
    g_K_eq = Equations('g_K = g_K_bar*n**4 : siemens/meter**2')
    g_Na_eq = Equations('g_Na = g_Na_bar*m**3*h : siemens/meter**2')

    # currents
    I_K_eq = Equations('I_K = g_K*(v - e_K) : ampere/meter**2')
    I_Na_eq = Equations('I_Na = g_Na*(v - e_Na) : ampere/meter**2')
    I_L_eq = Equations('I_L = g_L*(v - e_L) : ampere/meter**2')

    # external drive
    I_ext_eq = Equations('I_ext = I_stim(t) : ampere/meter**2')
```

```python
    # membrane
    membrane_eq = Equations('dv/dt = (I_ext - I_K - I_Na - I_L)/C_mem : volt')

    return [g_K_eq, g_Na_eq, I_K_eq, I_Na_eq, I_L_eq, I_ext_eq, membrane_eq]


def plot_tau(ax, parameters):
    """Plot gating variable time constants as function of membrane potential.

    ax -- matplotlib axes to be plotted on
    parameters -- dictionary of parameters for gating variable time constant equations
    """

    tau_group = NeuronGroup(100,
                            Equations('v : volt') +
                            reduce(operator.add, [construct_gating_variable_tau_
→equation(
                                gv) for gv in ['m', 'n', 'h']]),
                            method='euler', namespace=parameters)

    min_v = -100
    max_v = 100
    tau_group.v = np.linspace(min_v, max_v, len(tau_group))*mV

    ax.plot(tau_group.v/mV, tau_group.tau_m/ms, label=r'$\tau_m$')
    ax.plot(tau_group.v/mV, tau_group.tau_n/ms, label=r'$\tau_n$')
    ax.plot(tau_group.v/mV, tau_group.tau_h/ms, label=r'$\tau_h$')

    ax.set_xlabel('$v$ (mV)')
    ax.set_ylabel(r'$\tau$ (ms)')
    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()
    ax.legend()


def plot_inf(ax, parameters):
    """Plot gating variable steady-state values as function of membrane potential.

    ax -- matplotlib axes to be plotted on
    parameters -- dictionary of parameters for gating variable steady-state equations
    """

    inf_group = NeuronGroup(100,
                            Equations('v : volt') +
                            reduce(operator.add, [construct_gating_variable_inf_
→equation(
                                gv) for gv in ['m', 'n', 'h']]),
                            method='euler', namespace=parameters)
    inf_group.v = np.linspace(-100, 100, len(inf_group))*mV

    ax.plot(inf_group.v/mV, inf_group.m_inf, label=r'$m_\infty$')
    ax.plot(inf_group.v/mV, inf_group.n_inf, label=r'$n_\infty$')
    ax.plot(inf_group.v/mV, inf_group.h_inf, label=r'$h_\infty$')
    ax.set_xlabel('$v$ (mV)')
    ax.set_ylabel('steady-state activation')
    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()
```

```python
    ax.legend()


def plot_membrane_voltage(ax, statemon):
    """Plot simulation result: membrane potential.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with v recorded)
    """

    ax.plot(statemon.t/ms, statemon.v[0]/mV, label='membrane voltage')
    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel('$v$ (mV)')
    ax.axhline(0, linestyle='dashed')
    ax.legend()


def plot_gating_variable_activations(ax, statemon):
    """Plot simulation result: gating variables.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with m, n and h recorded)
    """

    ax.plot(statemon.t/ms, statemon.m[0], label='$m$')
    ax.plot(statemon.t/ms, statemon.n[0], label='$n$')
    ax.plot(statemon.t/ms, statemon.h[0], label='$h$')
    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel('activation')
    ax.legend()


def plot_conductances(ax, statemon):
    """Plot simulation result: conductances.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with g_K and g_Na recorded)
    """

    ax.plot(statemon.t/ms, statemon.g_K[0] / (mS/(cmeter**2)),
            label=r'$g_\mathregular{K}$')

    ax.plot(statemon.t/ms, statemon.g_Na[0] / (mS/(cmeter**2)),
            label=r'$g_\mathregular{Na}$')

    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel('$g$ (mS/cm$^2$)')
    ax.legend()


def plot_currents(ax, statemon):
    """Plot simulation result: currents.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with I_K, I_Na and I_L recorded)
    """
```

```python
    ax.plot(statemon.t/ms,
            statemon.I_K[0] / (uA/(cmeter**2)),
            label=r'$I_\mathregular{K}$')

    ax.plot(statemon.t/ms, statemon.I_Na[0] / (uA/(cmeter**2)),
            label=r'$I_\mathregular{Na}$')

    ax.plot(statemon.t/ms, (statemon.I_Na[0] + statemon.I_K[0] +
                            statemon.I_L[0]) / (uA/(cmeter**2)),
            label=r'$I_\mathregular{Na} + I_\mathregular{K} + I_\mathregular{L}$')

    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel(r'I ($\mu$A/cm$^2$)')
    ax.legend()


def plot_current_stimulus(ax, statemon):
    """Plot simulation result: external current stimulus.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with I_ext recorded)
    """

    ax.plot(statemon.t/ms, statemon.I_ext[0] /
            (uA/(cmeter**2)), label=r'$I_\mathregular{ext}$')

    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel(r'I ($\mu$A/cm$^2$)')
    ax.legend()


def plot_gating_variable_time_constants(ax, statemon):
    """Plot simulation result: gating variable time constants.

    ax -- matplotlib axes to be plotted on
    statemon -- StateMonitor (with tau_m, tau_n and tau_h recorded)
    """

    ax.plot(statemon.t/ms, statemon.tau_m[0]/ms, label=r'$\tau_m$')
    ax.plot(statemon.t/ms, statemon.tau_n[0]/ms, label=r'$\tau_n$')
    ax.plot(statemon.t/ms, statemon.tau_h[0]/ms, label=r'$\tau_h$')

    ax.set_xlabel('$t$ (ms)')
    ax.set_ylabel(r'$\tau$ (ms)')
    ax.legend()


def run_simulation(parameters):
    """Run the simulation.

    parameters -- dictionary with parameters
    """

    equations = []
    for gating_variable in ["m", "n", "h"]:
        equations.append(
            construct_gating_variable_inf_equation(gating_variable))
```

```python
        equations.append(
            construct_gating_variable_tau_equation(gating_variable))
        equations.append(construct_gating_variable_ode(gating_variable))
    equations += construct_neuron_ode()

    eqs_HH = reduce(operator.add, equations)
    group = NeuronGroup(1, eqs_HH, method='euler', namespace=parameters)


    group.v = parameters["v_initial"]

    group.m = parameters["m_initial"]
    group.n = parameters["n_initial"]
    group.h = parameters["h_initial"]

    statemon = StateMonitor(group, ['v',
                                    'I_ext',
                                    'm', 'n', 'h',
                                    'g_K', 'g_Na',
                                    'I_K', 'I_Na', 'I_L',
                                    'tau_m', 'tau_n', 'tau_h'],
                            record=True)


    defaultclock.dt = parameters["defaultclock_dt"]
    run(parameters["duration"])

    return statemon


def main(parameters):
    """Run simulation and return matplotlib figure.

    parameters -- dictionary with parameters
    """

    statemon = run_simulation(parameters)

    fig = plt.figure(figsize=(20, 15), constrained_layout=True)
    gs = fig.add_gridspec(6, 2)

    ax0 = fig.add_subplot(gs[0, 0])
    ax1 = fig.add_subplot(gs[1, 0])
    ax2 = fig.add_subplot(gs[2, 0])
    ax3 = fig.add_subplot(gs[3, 0])
    ax4 = fig.add_subplot(gs[4, 0])
    ax5 = fig.add_subplot(gs[5, 0])
    ax6 = fig.add_subplot(gs[:3, 1])
    ax7 = fig.add_subplot(gs[3:, 1])

    plot_membrane_voltage(ax0, statemon)
    plot_gating_variable_activations(ax1, statemon)
    plot_conductances(ax2, statemon)
    plot_currents(ax3, statemon)
    plot_current_stimulus(ax4, statemon)
    plot_gating_variable_time_constants(ax5, statemon)

    plot_tau(ax6, parameters)
    plot_inf(ax7, parameters)
```

```
    return fig


parameters = {

    # Boltzmann function parameters
    'v_n_half': 12*mV,
    'v_m_half': 25*mV,
    'v_h_half': 3*mV,

    'k_n': 15*mV,
    'k_m': 9*mV,
    'k_h': -7*mV,

    # Gaussian function parameters
    'v_n_max': -14*mV,
    'v_m_max': 27*mV,
    'v_h_max': -2*mV,

    'sigma_n': 50*mV,
    'sigma_m': 30*mV,
    'sigma_h': 20*mV,

    'c_n_amp': 4.7*ms,
    'c_m_amp': 0.46*ms,
    'c_h_amp': 7.4*ms,

    'c_n_base': 1.1*ms,
    'c_m_base': 0.04*ms,
    'c_h_base': 1.2*ms,

    # conductances
    'g_K_bar': 36*mS / (cmeter**2),
    'g_Na_bar': 120*mS / (cmeter**2),
    'g_L': 0.3*mS / (cmeter**2),

    # reversal potentials
    'e_K': -12*mV,
    'e_Na': 120*mV,
    'e_L': 10.6*mV,

    # membrane capacitance
    'C_mem': 1*uF / cmeter**2,

    # initial membrane voltage
    'v_initial': 0*mV,

    # initial gating variable activations
    'm_initial': 0.05,
    'n_initial': 0.32,
    'h_initial': 0.60,

    # external stimulus at 2 ms with 4 uA/cm^2 and at 10 ms with 15 uA/cm^2
    # for 0.5 ms each
    'I_stim': TimedArray(values=([0]*4+[4]+[0]*15+[15]+[0])*uA/(cmeter**2),
                         dt=0.5*ms),
```

```python
    # simulation time step
    'defaultclock_dt': 0.01*ms,

    # simulation duration
    'duration': 20*ms
}

linestyle_cycler = cycler('linestyle',['-','--',':','-.'])
plt.rc('axes', prop_cycle=linestyle_cycler)

fig = main(parameters)

plt.show()
```
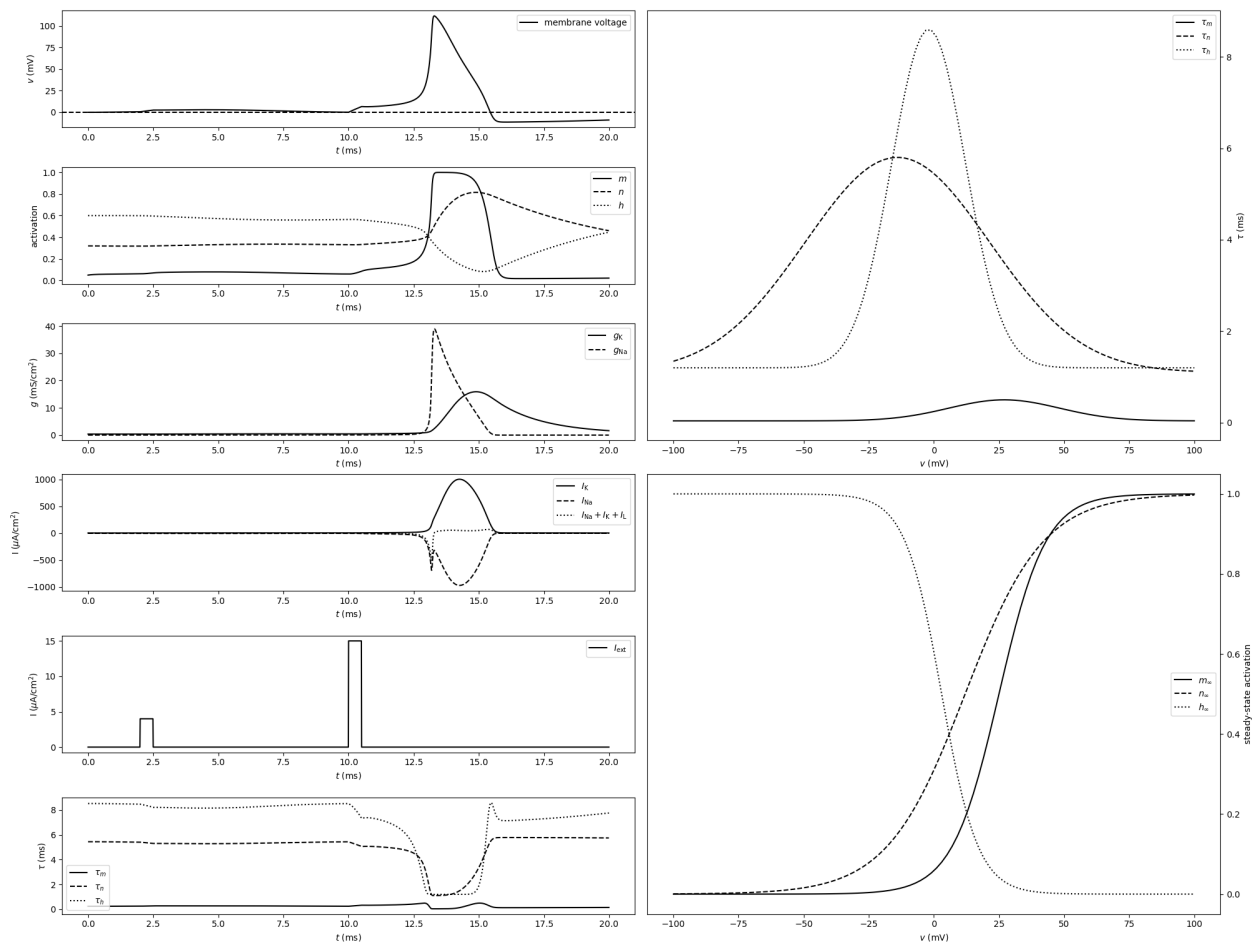
5.9.2 Example: compare_GSL_to_conventional

Example using GSL ODE solvers with a variable time step and comparing it to the Brian solver.

For highly accurate simulations, i.e. simulations with a very low desired error, the GSL simulation with a variable time step can be faster because it uses a low time step only when it is necessary. In biologically detailed models (e.g. of the Hodgkin-Huxley type), the relevant time constants are very short around an action potential, but much longer when the neuron is near its resting potential. The following example uses a very simple neuron model (leaky integrate-and-fire), but simulates a change in relevant time constants by changing the actual time constant every 10ms, independently for each of 100 neurons. To accurately simulate this model with a fixed time step, the time step has to be very small, wasting many unnecessary steps for all the neurons where the time constant is long.

Note that using the GSL ODE solver is much slower, if both methods use a comparable number of steps, i.e. if the desired accuracy is low enough so that a single step per "Brian time step" is enough.

```python
from brian2 import *
import time

# Run settings
start_dt = .1 * ms
method = 'rk2'
error = 1.e-6  # requested accuracy


def runner(method, dt, options=None):
    seed(0)
    I = 5
    group = NeuronGroup(100, '''dv/dt = (-v + I)/tau : 1
                                tau : second''',
                        method=method,
                        method_options=options,
                        dt=dt)
    group.run_regularly('''v = rand()
                           tau = 0.1*ms + rand()*9.9*ms''', dt=10*ms)

    rec_vars = ['v', 'tau']
    if 'gsl' in method:
        rec_vars += ['_step_count']
    net = Network(group)
    net.run(0 * ms)
    mon = StateMonitor(group, rec_vars, record=True, dt=start_dt)
    net.add(mon)
    start = time.time()
    net.run(1 * second)
    mon.add_attribute('run_time')
    mon.run_time = time.time() - start
    return mon


lin = runner('linear', start_dt)
method_options = {'save_step_count': True,
                  'absolute_error': error,
                  'max_steps': 10000}
gsl = runner('gsl_%s' % method, start_dt, options=method_options)

print("Running with GSL integrator and variable time step:")
```

(continues on next page)

```python
print('Run time: %.3fs' % gsl.run_time)

# check gsl error
assert np.max(np.abs(
    lin.v - gsl.v)) < error, "Maximum error gsl integration too large: %f" % np.max(
    np.abs(lin.v - gsl.v))
print("average step count: %.1f" % np.mean(gsl._step_count))
print("average absolute error: %g" % np.mean(np.abs(gsl.v - lin.v)))

print("\nRunning with exact integration and fixed time step:")
dt = start_dt
count = 0
dts = []
avg_errors = []
max_errors = []
runtimes = []
while True:
    print('Using dt: %s' % str(dt))
    brian = runner(method, dt)
    print('\tRun time: %.3fs' % brian.run_time)
    avg_errors.append(np.mean(np.abs(brian.v - lin.v)))
    max_errors.append(np.max(np.abs(brian.v - lin.v)))
    dts.append(dt)
    runtimes.append(brian.run_time)
    if np.max(np.abs(brian.v - lin.v)) > error:
        print('\tError too high (%g), decreasing dt' % np.max(
            np.abs(brian.v - lin.v)))
        dt *= .5
        count += 1
    else:
        break
print("Desired error level achieved:")
print("average step count: %.2fs" % (start_dt / dt))
print("average absolute error: %g" % np.mean(np.abs(brian.v - lin.v)))

print('Run time: %.3fs' % brian.run_time)
if brian.run_time > gsl.run_time:
    print("This is %.1f times slower than the simulation with GSL's variable "
          "time step method." % (brian.run_time / gsl.run_time))
else:
    print("This is %.1f times faster than the simulation with GSL's variable "
          "time step method." % (gsl.run_time / brian.run_time))

fig, (ax1, ax2) = plt.subplots(1, 2)
ax2.axvline(1e-6, color='gray')
for label, gsl_error, std_errors, ax in [('average absolute error', np.mean(np.
→abs(gsl.v - lin.v)), avg_errors, ax1),
                                         ('maximum absolute error', np.max(np.abs(gsl.
→v - lin.v)), max_errors, ax2)]:
    ax.set(xscale='log', yscale='log')
    ax.plot([], [], 'o', color='C0', label='fixed time step')  # for the legend entry
    for (error, runtime, dt) in zip(std_errors, runtimes, dts):
        ax.plot(error, runtime, 'o', color='C0')
        ax.annotate('%s' % str(dt), xy=(error, runtime), xytext=(2.5, 5),
                    textcoords='offset points', color='C0')
    ax.plot(gsl_error, gsl.run_time, 'o', color='C1', label='variable time step (GSL)
→')
```

```
    ax.set(xlabel=label, xlim=(10**-10, 10**1))
ax1.set_ylabel('runtime (s)')
ax2.legend(loc='lower left')

plt.show()
```
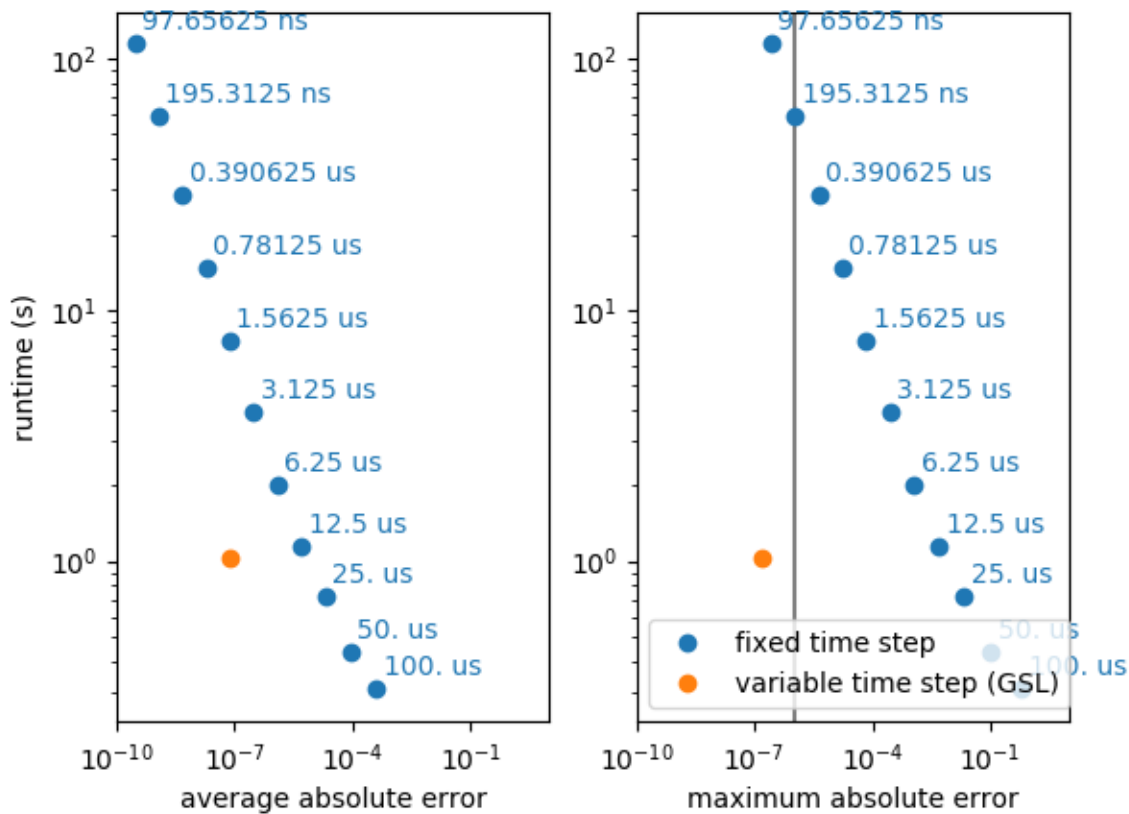

5.9.3 Example: custom_events

Example demonstrating the use of custom events.

Here we have three neurons, the first is Poisson spiking and connects to neuron G, which in turn connects to neuron H. Neuron G has two variables v and g, and the incoming Poisson spikes cause an instantaneous increase in variable g. g decays rapidly, and in turn causes a slow increase in v. If v crosses a threshold, it causes a standard spike and reset. If g crosses a threshold, it causes a custom event `gspike`, and if it returns below that threshold it causes a custom event `end_gspike`. The standard spike event when v crosses a threshold causes an instantaneous increase in variable x in neuron H (which happens through the standard `pre` pathway in the synapses), and the gspike event causes an increase in variable y (which happens through the custom pathway `gpath`).

```
from brian2 import *
# Input Poisson spikes
inp = PoissonGroup(1, rates=250*Hz)
```

```python
# First group G
eqs_G = '''
dv/dt = (g-v)/(50*ms) : 1
dg/dt = -g/(10*ms) : 1
allow_gspike : boolean
'''
G = NeuronGroup(1, eqs_G, threshold='v>1',
                reset='v = 0; g = 0; allow_gspike = True;',
                events={'gspike': 'g>1 and allow_gspike',
                        'end_gspike': 'g<1 and not allow_gspike'})
G.run_on_event('gspike', 'allow_gspike = False')
G.run_on_event('end_gspike', 'allow_gspike = True')
# Second group H
eqs_H = '''
dx/dt = -x/(10*ms) : 1
dy/dt = -y/(10*ms) : 1
'''
H = NeuronGroup(1, eqs_H)
# Synapses from input Poisson group to G
Sin = Synapses(inp, G, on_pre='g += 0.5')
Sin.connect()
# Synapses from G to H
S = Synapses(G, H,
             on_pre={'pre': 'x += 1',
                     'gpath': 'y += 1'},
             on_event={'pre': 'spike',
                       'gpath': 'gspike'})
S.connect()
# Monitors
Mstate = StateMonitor(G, ('v', 'g'), record=True)
Mgspike = EventMonitor(G, 'gspike', 'g')
Mspike = SpikeMonitor(G, 'v')
MHstate = StateMonitor(H, ('x', 'y'), record=True)
# Initialise and run
G.allow_gspike = True
run(500*ms)
# Plot
figure(figsize=(10, 4))
subplot(121)
plot(Mstate.t/ms, Mstate.g[0], '-g', label='g')
plot(Mstate.t/ms, Mstate.v[0], '-b', lw=2, label='V')
plot(Mspike.t/ms, Mspike.v, 'ob', label='_nolegend_')
plot(Mgspike.t/ms, Mgspike.g, 'og', label='_nolegend_')
xlabel('Time (ms)')
title('Presynaptic group G')
legend(loc='best')
subplot(122)
plot(MHstate.t/ms, MHstate.y[0], '-r', label='y')
plot(MHstate.t/ms, MHstate.x[0], '-k', lw=2, label='x')
xlabel('Time (ms)')
title('Postsynaptic group H')
legend(loc='best')
tight_layout()
show()
```
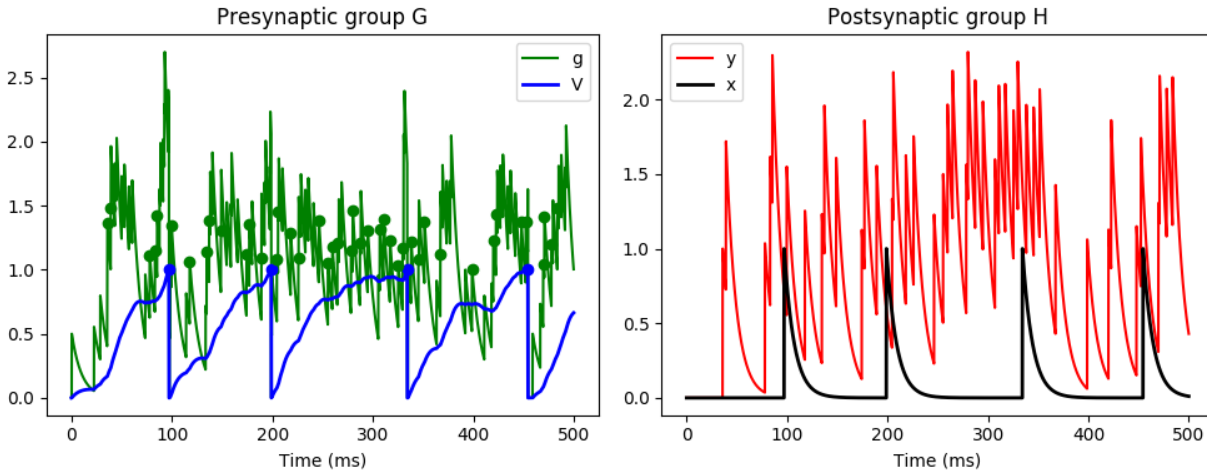
5.9.4 Example: exprel_function

Show the improved numerical accuracy when using the `exprel()` function in rate equations.

Rate equations for channel opening/closing rates often include a term of the form $\frac{x}{\exp(x)-1}$. This term is problematic for two reasons:

- It is not defined for $x = 0$ (where it should equal to 1 for continuity);

- For values $x \approx 0$, there is a loss of accuracy.

For better accuracy, and to avoid issues at $x = 0$, Brian provides the function `exprel()`, which is equivalent to $\frac{\exp(x)-1}{x}$, but with better accuracy and the expected result at $x = 0$. In this example, we demonstrate the advantage of expressing a typical rate equation from the HH model with `exprel()`.

```python
from brian2 import *

# Dummy group to evaluate the rate equation at various points
eqs = '''v : volt
         # opening rate from the HH model
         alpha_simple = 0.32*(mV**-1)*(-50*mV-v)/
                        (exp((-50*mV-v)/(4*mV))-1.)/ms : Hz
         alpha_improved = 0.32*(mV**-1)*4*mV/exprel((-50*mV-v)/(4*mV))/ms : Hz'''
neuron = NeuronGroup(1000, eqs)

# Use voltage values around the problematic point
neuron.v = np.linspace(-50 - .5e-6, -50 + .5e-6, len(neuron))*mV

fig, ax = plt.subplots()
ax.plot((neuron.v + 50*mV)/nvolt, neuron.alpha_simple,
        '.', label=r'$\alpha_\mathrm{simple}$')
ax.plot((neuron.v + 50*mV)/nvolt, neuron.alpha_improved,
        'k', label=r'$\alpha_\mathrm{improved}$')
ax.legend()
ax.set(xlabel='$v$ relative to -50mV (nV)', ylabel=r'$\alpha$ (Hz)')
ax.ticklabel_format(useOffset=False)
plt.tight_layout()
plt.show()
```
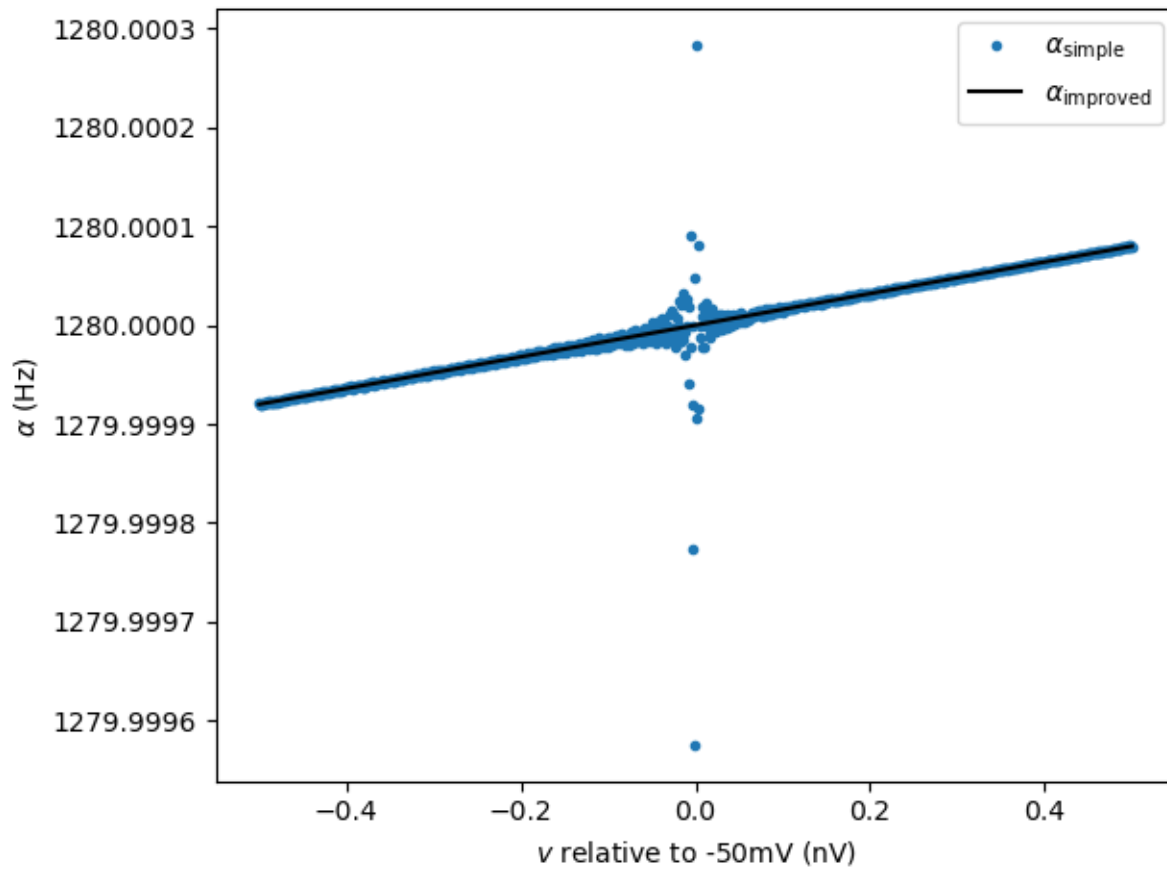
5.9.5 Example: float_32_64_benchmark

Benchmark showing the performance of float32 versus float64.

```python
from brian2 import *
from brian2.devices.device import reset_device, reinit_devices

# CUBA benchmark
def run_benchmark(name):
    if name=='CUBA':

        taum = 20*ms
        taue = 5*ms
        taui = 10*ms
        Vt = -50*mV
        Vr = -60*mV
        El = -49*mV

        eqs = '''
        dv/dt  = (ge+gi-(v-El))/taum : volt (unless refractory)
        dge/dt = -ge/taue : volt
        dgi/dt = -gi/taui : volt
```

(continues on next page)

```
    '''

    P = NeuronGroup(4000, eqs, threshold='v>Vt', reset='v = Vr', refractory=5*ms,
                    method='exact')
    P.v = 'Vr + rand() * (Vt - Vr)'
    P.ge = 0*mV
    P.gi = 0*mV

    we = (60*0.27/10)*mV # excitatory synaptic weight (voltage)
    wi = (-20*4.5/10)*mV # inhibitory synaptic weight
    Ce = Synapses(P, P, on_pre='ge += we')
    Ci = Synapses(P, P, on_pre='gi += wi')
    Ce.connect('i<3200', p=0.02)
    Ci.connect('i>=3200', p=0.02)

elif name=='COBA':

    # Parameters
    area = 20000 * umetre ** 2
    Cm = (1 * ufarad * cm ** -2) * area
    gl = (5e-5 * siemens * cm ** -2) * area

    El = -60 * mV
    EK = -90 * mV
    ENa = 50 * mV
    g_na = (100 * msiemens * cm ** -2) * area
    g_kd = (30 * msiemens * cm ** -2) * area
    VT = -63 * mV
    # Time constants
    taue = 5 * ms
    taui = 10 * ms
    # Reversal potentials
    Ee = 0 * mV
    Ei = -80 * mV
    we = 6 * nS  # excitatory synaptic weight
    wi = 67 * nS  # inhibitory synaptic weight

    # The model
    eqs = Equations('''
    dv/dt = (gl*(El-v)+ge*(Ee-v)+gi*(Ei-v)-
             g_na*(m*m*m)*h*(v-ENa)-
             g_kd*(n*n*n*n)*(v-EK))/Cm : volt
    dm/dt = alpha_m*(1-m)-beta_m*m : 1
    dn/dt = alpha_n*(1-n)-beta_n*n : 1
    dh/dt = alpha_h*(1-h)-beta_h*h : 1
    dge/dt = -ge*(1./taue) : siemens
    dgi/dt = -gi*(1./taui) : siemens
    alpha_m = 0.32*(mV**-1)*4*mV/exprel((13*mV-v+VT)/(4*mV))/ms : Hz
    beta_m = 0.28*(mV**-1)*5*mV/exprel((v-VT-40*mV)/(5*mV))/ms : Hz
    alpha_h = 0.128*exp((17*mV-v+VT)/(18*mV))/ms : Hz
    beta_h = 4./(1+exp((40*mV-v+VT)/(5*mV)))/ms : Hz
    alpha_n = 0.032*(mV**-1)*5*mV/exprel((15*mV-v+VT)/(5*mV))/ms : Hz
    beta_n = .5*exp((10*mV-v+VT)/(40*mV))/ms : Hz
    ''')

    P = NeuronGroup(4000, model=eqs, threshold='v>-20*mV', refractory=3 * ms,
                    method='exponential_euler')
```

```python
        Pe = P[:3200]
        Pi = P[3200:]
        Ce = Synapses(Pe, P, on_pre='ge+=we')
        Ci = Synapses(Pi, P, on_pre='gi+=wi')
        Ce.connect(p=0.02)
        Ci.connect(p=0.02)

        # Initialization
        P.v = 'El + (randn() * 5 - 5)*mV'
        P.ge = '(randn() * 1.5 + 4) * 10.*nS'
        P.gi = '(randn() * 12 + 20) * 10.*nS'

    run(1 * second, profile=True)

    return sum(t for name, t in magic_network.profiling_info)

def generate_results(num_repeats):
    results = {}

    for name in ['CUBA', 'COBA']:
        for target in ['numpy', 'cython']:
            for dtype in [float32, float64]:
                prefs.codegen.target = target
                prefs.core.default_float_dtype = dtype
                times = [run_benchmark(name) for repeat in range(num_repeats)]
                results[name, target, dtype.__name__] = amin(times)

    for name in ['CUBA', 'COBA']:
        for dtype in [float32, float64]:
            times = []
            for _ in range(num_repeats):
                reset_device()
                reinit_devices()
                set_device('cpp_standalone', directory=None, with_output=False)
                prefs.core.default_float_dtype = dtype
                times.append(run_benchmark(name))
            results[name, 'cpp_standalone', dtype.__name__] = amin(times)

    return results

results = generate_results(3)

bar_width = 0.9
names = ['CUBA', 'COBA']
targets = ['numpy', 'cython', 'cpp_standalone']
precisions = ['float32', 'float64']

figure(figsize=(8, 8))
for j, name in enumerate(names):
    subplot(2, 2, 1+2*j)
    title(name)
    index = arange(len(targets))
    for i, precision in enumerate(precisions):
        bar(index+i*bar_width/len(precisions),
            [results[name, target, precision] for target in targets],
            bar_width/len(precisions), label=precision, align='edge')
    ylabel('Time (s)')
```

```python
    if j:
        xticks(index+0.5*bar_width, targets, rotation=45)
    else:
        xticks(index+0.5*bar_width, ('',)*len(targets))
        legend(loc='best')

    subplot(2, 2, 2+2*j)
    index = arange(len(precisions))
    for i, target in enumerate(targets):
        bar(index+i*bar_width/len(targets),
            [results[name, target, precision] for precision in precisions],
            bar_width/len(targets), label=target, align='edge')
    ylabel('Time (s)')
    if j:
        xticks(index+0.5*bar_width, precisions, rotation=45)
    else:
        xticks(index+0.5*bar_width, ('',)*len(precisions))
        legend(loc='best')

tight_layout()
show()
```

5.9.6 Example: modelfitting_sbi

Model fitting with simulation-based inference

In this example, a HH-type model is used to demonstrate simulation-based inference with the sbi toolbox (https://www.mackelab.org/sbi/). It is based on a fake current-clamp recording generated from the same model that we use in the inference process. Two of the parameters (the maximum sodium and potassium conductances) are considered parameters of the model.

For more details about this approach, see the references below.

To run this example, you need to install the sbi package, e.g. with:

```
pip install sbi
```

References:

- https://www.mackelab.org/sbi

- Tejero-Cantero et al., (2020). sbi: A toolkit for simulation-based inference. Journal of Open Source Software, 5(52), 2505, https://doi.org/10.21105/joss.02505

```python
import matplotlib.pyplot as plt

from brian2 import *
import sbi.utils
import sbi.analysis
import sbi.inference
import torch  # PyTorch

defaultclock.dt = 0.05*ms
```

```python
def simulate(params, I=1*nA, t_on=50*ms, t_total=350*ms):
    """
    Simulates the HH-model with Brian2 for parameter sets in params and the
    given input current (injection of I between t_on and t_total-t_on).

    Returns a dictionary {'t': time steps, 'v': voltage,
                          'I_inj': current, 'spike_count': spike count}.
    """
    assert t_total > 2*t_on
    t_off = t_total - t_on

    params = np.atleast_2d(params)
    # fixed parameters
    gleak = 10*nS
    Eleak = -70*mV
    VT = -60.0*mV
    C = 200*pF
    ENa = 53*mV
    EK = -107*mV

    # The conductance-based model
    eqs = '''
        dVm/dt = -(gNa*m**3*h*(Vm - ENa) + gK*n**4*(Vm - EK) + gleak*(Vm - Eleak) -
→I_inj) / C : volt
        I_inj = int(t >= t_on and t < t_off)*I : amp (shared)
        dm/dt = alpham*(1-m) - betam*m : 1
        dn/dt = alphan*(1-n) - betan*n : 1
        dh/dt = alphah*(1-h) - betah*h : 1

        alpham = (-0.32/mV) * (Vm - VT - 13.*mV) / (exp((-(Vm - VT - 13.*mV))/(4.
→*mV)) - 1)/ms : Hz
        betam = (0.28/mV) * (Vm - VT - 40.*mV) / (exp((Vm - VT - 40.*mV)/(5.*mV)) -
→1)/ms : Hz

        alphah = 0.128 * exp(-(Vm - VT - 17.*mV) / (18.*mV))/ms : Hz
        betah = 4/(1 + exp((-(Vm - VT - 40.*mV)) / (5.*mV)))/ms : Hz

        alphan = (-0.032/mV) * (Vm - VT - 15.*mV) / (exp((-(Vm - VT - 15.*mV)) / (5.
→*mV)) - 1)/ms : Hz
        betan = 0.5*exp(-(Vm - VT - 10.*mV) / (40.*mV))/ms : Hz
        # The parameters to fit
        gNa : siemens (constant)
        gK : siemens (constant)
        '''
    neurons = NeuronGroup(params.shape[0], eqs, threshold='m>0.5', refractory='m>0.5',
                          method='exponential_euler', name='neurons')
    Vm_mon = StateMonitor(neurons, 'Vm', record=True, name='Vm_mon')
    spike_mon = SpikeMonitor(neurons, record=False, name='spike_mon')  #record=False
→ do not record times
    neurons.gNa_ = params[:, 0]*uS
    neurons.gK = params[:, 1]*uS

    neurons.Vm = 'Eleak'
    neurons.m = '1/(1 + betam/alpham)'        # Would be the solution when dm/dt = 0
    neurons.h = '1/(1 + betah/alphah)'        # Would be the solution when dh/dt = 0
    neurons.n = '1/(1 + betan/alphan)'        # Would be the solution when dn/dt = 0
```

```python
    run(t_total)
    # For convenient plotting, reconstruct the current
    I_inj = ((Vm_mon.t >= t_on) & (Vm_mon.t < t_off))*I
    return dict(v=Vm_mon.Vm,
                t=Vm_mon.t,
                I_inj=I_inj,
                spike_count=spike_mon.count)


def calculate_summary_statistics(x):
    """Calculate summary statistics for results in x"""
    I_inj = x["I_inj"]
    v = x["v"]/mV

    spike_count = x["spike_count"]
    # Mean and standard deviation during stimulation
    v_active = v[:, I_inj > 0*nA]
    mean_active = np.mean(v_active, axis=1)
    std_active = np.std(v_active, axis=1)
    # Height of action potential peaks
    max_v = np.max(v_active, axis=1)

    # concatenation of summary statistics
    sum_stats = np.vstack((spike_count, mean_active, std_active, max_v))

    return sum_stats.T


def simulation_wrapper(params):
    """
    Returns summary statistics from conductance values in `params`.
    Summarizes the output of the simulation and converts it to `torch.Tensor`.
    """
    obs = simulate(params)
    summstats = torch.as_tensor(calculate_summary_statistics(obs))
    return summstats.to(torch.float32)


if __name__ == '__main__':
    # Define prior distribution over parameters
    prior_min = [.5, 1e-4]  # (gNa, gK) in µS
    prior_max = [80.,15.]
    prior = sbi.utils.torchutils.BoxUniform(low=torch.as_tensor(prior_min),
                                            high=torch.as_tensor(prior_max))

    # Simulate samples from the prior distribution
    theta = prior.sample((10_000,))
    print('Simulating samples from prior simulation... ', end='')
    stats = simulation_wrapper(theta.numpy())
    print('done.')

    # Train inference network
    density_estimator_build_fun = sbi.utils.posterior_nn(model='mdn')
    inference = sbi.inference.SNPE(prior,
                                   density_estimator=density_estimator_build_fun)
    print('Training inference network... ')
```

```python
inference.append_simulations(theta, stats).train()
posterior = inference.build_posterior()

# true parameters for real ground truth data
true_params = np.array([[32., 1.]])
true_data = simulate(true_params)
t = true_data['t']
I_inj = true_data['I_inj']
v = true_data['v']
xo = calculate_summary_statistics(true_data)
print("The true summary statistics are:  ", xo)

# Plot estimated posterior distribution
samples = posterior.sample((1000,), x=xo, show_progress_bars=False)
labels_params = [r'$\overline{g}_{Na}$', r'$\overline{g}_{K}$']
sbi.analysis.pairplot(samples,
                      limits=[[.5, 80], [1e-4, 15.]],
                      ticks=[[.5, 80], [1e-4, 15.]],
                      figsize=(4, 4),
                      points=true_params, labels=labels_params,
                      points_offdiag={'markersize': 6},
                      points_colors=['r'])
plt.tight_layout()

# Draw a single sample from the posterior and convert to numpy for plotting.
posterior_sample = posterior.sample((1,), x=xo,
                                    show_progress_bars=False).numpy()
x = simulate(posterior_sample)

# plot observation and sample
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(t/ms, v[0, :]/mV, lw=2, label='observation')
ax.plot(t/ms, x['v'][0, :]/mV, '--', lw=2, label='posterior sample')
ax.legend()
ax.set(xlabel='time (ms)', ylabel='voltage (mV)')
plt.show()
```
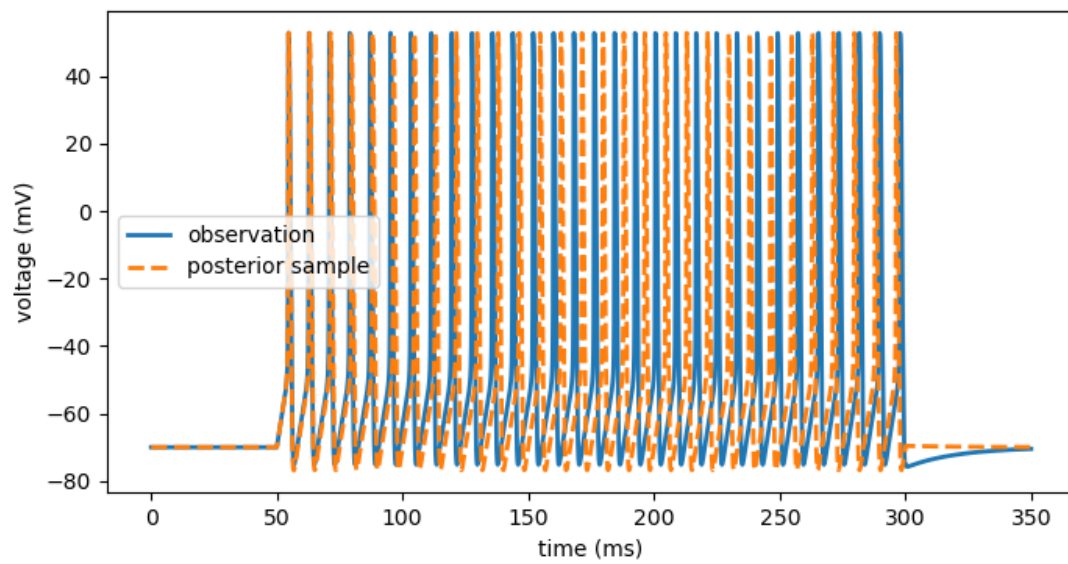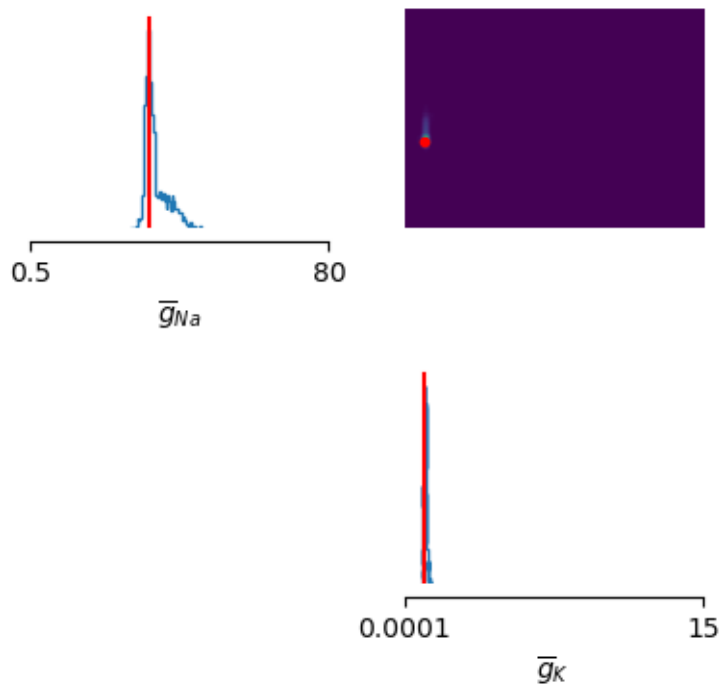
5.9.7 Example: opencv_movie

An example that uses a function from external C library (OpenCV in this case). Works for all C-based code generation targets (i.e. for cython and cpp_standalone device) and for numpy (using the Python bindings).

This example needs a working installation of OpenCV 3.x and its Python bindings. It has been tested on 64 bit Linux in a conda environment with packages from the `conda-forge` channels (opencv 3.4.4, x264 1!152.20180717, ffmpeg 4.1).

```python
import os
import urllib.request, urllib.error, urllib.parse
import cv2  # Import OpenCV2

from brian2 import *

defaultclock.dt = 1*ms
prefs.codegen.target = 'cython'
prefs.logging.std_redirection = False
set_device('cpp_standalone', clean=True)
filename = os.path.abspath('Megamind.avi')

if not os.path.exists(filename):
    print('Downloading the example video file')
    response = urllib.request.urlopen('http://docs.opencv.org/2.4/_downloads/Megamind.
↪avi')
    data = response.read()
    with open(filename, 'wb') as f:
        f.write(data)

video = cv2.VideoCapture(filename)
width, height, frame_count = (int(video.get(cv2.CAP_PROP_FRAME_WIDTH)),
                             int(video.get(cv2.CAP_PROP_FRAME_HEIGHT)),
                             int(video.get(cv2.CAP_PROP_FRAME_COUNT)))
fps = 24
time_between_frames = 1*second/fps

@implementation('cpp', '''
double* get_frame(bool new_frame)
{
    // The following initializations will only be executed once
    static cv::VideoCapture source("VIDEO_FILENAME");
    static cv::Mat frame;
    static double* grayscale_frame = (double*)malloc(VIDEO_WIDTH*VIDEO_
↪HEIGHT*sizeof(double));
    if (new_frame)
    {
        source >> frame;
        double mean_value = 0;
        for (int row=0; row<VIDEO_HEIGHT; row++)
            for (int col=0; col<VIDEO_WIDTH; col++)
            {
                const double grayscale_value = (frame.at<cv::Vec3b>(row, col)[0] +
                                                frame.at<cv::Vec3b>(row, col)[1] +
                                                frame.at<cv::Vec3b>(row, col)[2])/(3.
↪0*128);
                mean_value += grayscale_value / (VIDEO_WIDTH * VIDEO_HEIGHT);
```

(continues on next page)

```
                    grayscale_frame[row*VIDEO_WIDTH + col] = grayscale_value;
            }
        // subtract the mean
        for (int i=0; i<VIDEO_HEIGHT*VIDEO_WIDTH; i++)
            grayscale_frame[i] -= mean_value;
    }
    return grayscale_frame;
}

double video_input(const int x, const int y)
{
    // Get the current frame (or a new frame in case we are asked for the first
    // element
    double *frame = get_frame(x==0 && y==0);
    return frame[y*VIDEO_WIDTH + x];
}
'''.replace('VIDEO_FILENAME', filename),
               libraries=['opencv_core',
                          'opencv_highgui',
                          'opencv_videoio'],
               headers=['<opencv2/core/core.hpp>',
                        '<opencv2/highgui/highgui.hpp>'],
               define_macros=[('VIDEO_WIDTH', width),
                              ('VIDEO_HEIGHT', height)])
@check_units(x=1, y=1, result=1)
def video_input(x, y):
    # we assume this will only be called in the custom operation (and not for
    # example in a reset or synaptic statement), so we don't need to do indexing
    # but we can directly return the full result
    _, frame = video.read()
    grayscale = frame.mean(axis=2)
    grayscale /= 128.  # scale everything between 0 and 2
    return grayscale.ravel() - grayscale.ravel().mean()


N = width * height
tau, tau_th = 10*ms, time_between_frames
G = NeuronGroup(N, '''dv/dt = (-v + I)/tau : 1
                      dv_th/dt = -v_th/tau_th : 1
                      row : integer (constant)
                      column : integer (constant)
                      I : 1 # input current''',
                threshold='v>v_th', reset='v=0; v_th = 3*v_th + 1.0',
                method='exact')
G.v_th = 1
G.row = 'i//width'
G.column = 'i%width'

G.run_regularly('I = video_input(column, row)',
                dt=time_between_frames)
mon = SpikeMonitor(G)
runtime = frame_count*time_between_frames
run(runtime, report='text')

# Avoid going through the whole Brian2 indexing machinery too much
i, t, row, column = mon.i[:], mon.t[:], G.row[:], G.column[:]
```

```python
import matplotlib.animation as animation

# TODO: Use overlapping windows
stepsize = 100*ms
def next_spikes():
    step = next_spikes.step
    if step*stepsize > runtime:
        next_spikes.step=0
        raise StopIteration()
    spikes = i[(t>=step*stepsize) & (t<(step+1)*stepsize)]
    next_spikes.step += 1
    yield column[spikes], row[spikes]
next_spikes.step = 0

fig, ax = plt.subplots()
dots, = ax.plot([], [], 'k.', markersize=2, alpha=.25)
ax.set_xlim(0, width)
ax.set_ylim(0, height)
ax.invert_yaxis()
def run(data):
    x, y = data
    dots.set_data(x, y)

ani = animation.FuncAnimation(fig, run, next_spikes, blit=False, repeat=True,
                              repeat_delay=1000)
plt.show()
```

5.9.8 Example: stochastic_odes

Demonstrate the correctness of the "derivative-free Milstein method" for multiplicative noise.

```python
from brian2 import *
# We only get exactly the same random numbers for the exact solution and the
# simulation if we use the numpy code generation target
prefs.codegen.target = 'numpy'

# setting a random seed makes all variants use exactly the same Wiener process
seed = 12347

X0 = 1
mu = 0.5/second # drift
sigma = 0.1/second #diffusion

runtime = 1*second


def simulate(method, dt):
    """
    simulate geometrical Brownian with the given method
    """
    np.random.seed(seed)
    G = NeuronGroup(1, 'dX/dt = (mu - 0.5*second*sigma**2)*X + X*sigma*xi*second**.5:␣
↪1',
```

```python
                       dt=dt, method=method)
    G.X = X0
    mon = StateMonitor(G, 'X', record=True)
    net = Network(G, mon)
    net.run(runtime)
    return mon.t_[:], mon.X.flatten()


def exact_solution(t, dt):
    """
    Return the exact solution for geometrical Brownian motion at the given
    time points
    """
    # Remove units for simplicity
    my_mu = float(mu)
    my_sigma = float(sigma)
    dt = float(dt)
    t = asarray(t)

    np.random.seed(seed)
    # We are calculating the values at the *start* of a time step, as when using
    # a StateMonitor. Therefore the Brownian motion starts with zero
    brownian = np.hstack([0, cumsum(sqrt(dt) * np.random.randn(len(t)-1))])

    return (X0 * exp((my_mu - 0.5*my_sigma**2)*(t+dt) + my_sigma*brownian))

figure(1, figsize=(16, 7))
figure(2, figsize=(16, 7))

methods = ['milstein', 'heun']
dts = [1*ms, 0.5*ms, 0.2*ms, 0.1*ms, 0.05*ms, 0.025*ms, 0.01*ms, 0.005*ms]

rows = floor(sqrt(len(dts)))
cols = ceil(1.0 * len(dts) / rows)
errors = dict([(method, zeros(len(dts))) for method in methods])
for dt_idx, dt in enumerate(dts):
    print('dt: %s' % dt)
    trajectories = {}
    # Test the numerical methods
    for method in methods:
        t, trajectories[method] = simulate(method, dt)
    # Calculate the exact solution
    exact = exact_solution(t, dt)

    for method in methods:
        # plot the trajectories
        figure(1)
        subplot(rows, cols, dt_idx+1)
        plot(t, trajectories[method], label=method, alpha=0.75)

        # determine the mean absolute error
        errors[method][dt_idx] = mean(abs(trajectories[method] - exact))
        # plot the difference to the real trajectory
        figure(2)
        subplot(rows, cols, dt_idx+1)
        plot(t, trajectories[method] - exact, label=method, alpha=0.75)
```

```python
    figure(1)
    plot(t, exact, color='gray', lw=2, label='exact', alpha=0.75)
    title('dt = %s' % str(dt))
    xticks([])

figure(1)
legend(frameon=False, loc='best')
tight_layout()

figure(2)
legend(frameon=False, loc='best')
tight_layout()

figure(3)
for method in methods:
    plot(array(dts) / ms, errors[method], 'o', label=method)
legend(frameon=False, loc='best')
xscale('log')
yscale('log')
xlabel('dt (ms)')
ylabel('Mean absolute error')
tight_layout()

show()
```
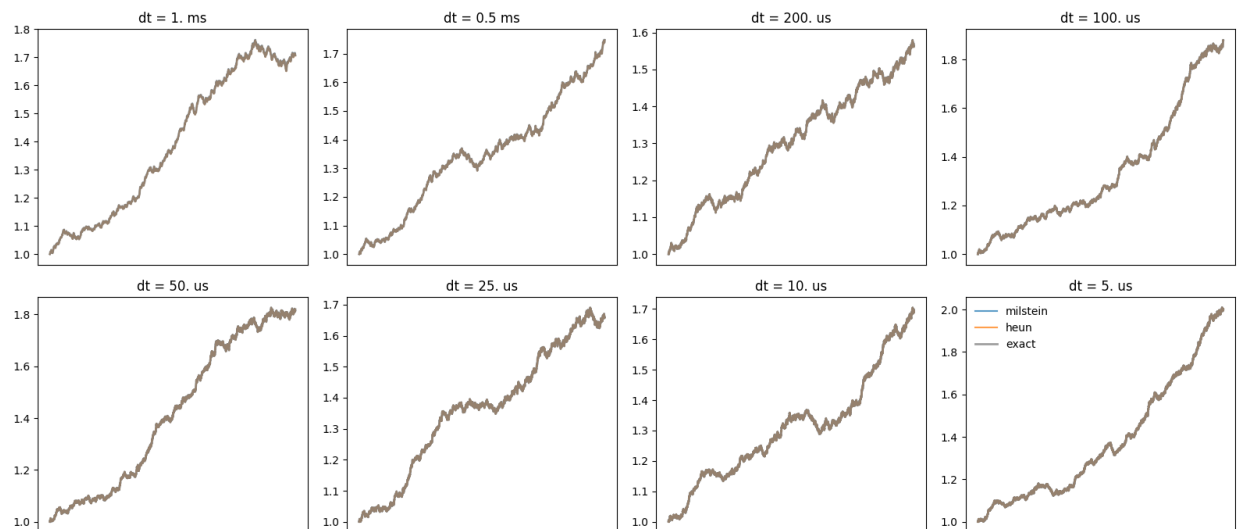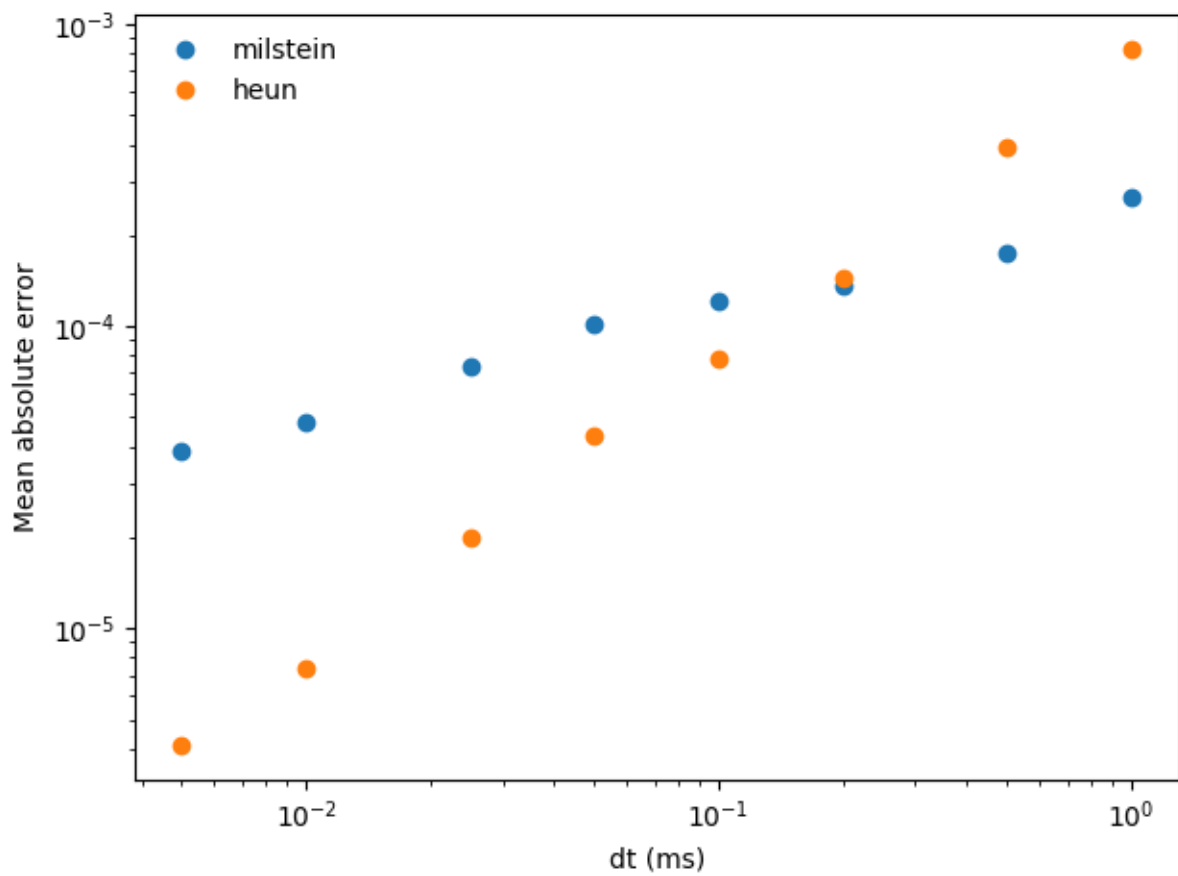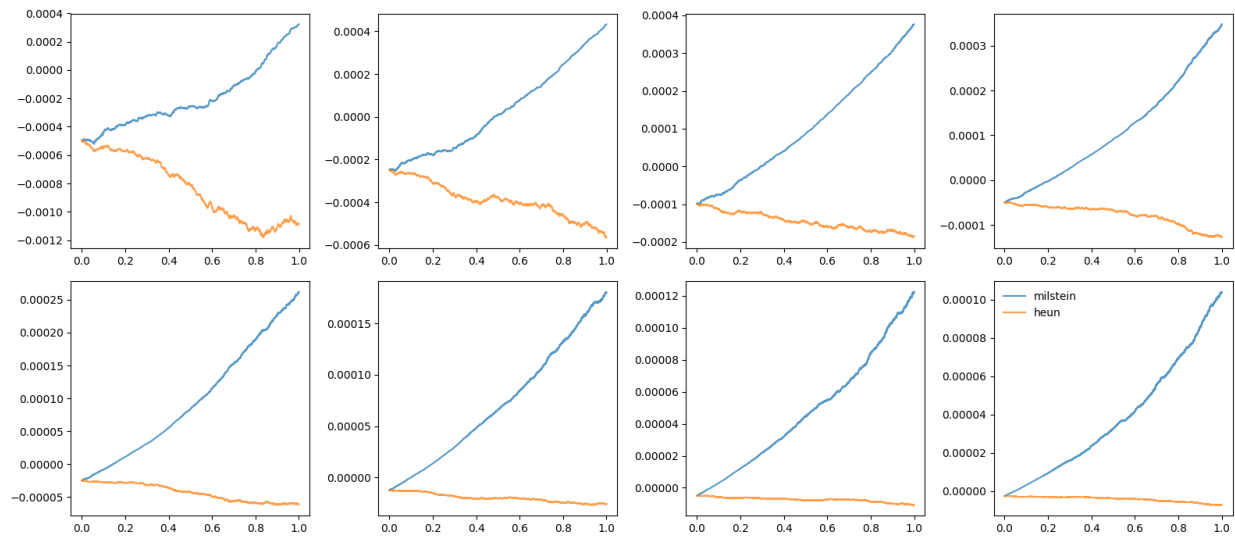
5.10 compartmental

5.10.1 Example: bipolar_cell

A pseudo MSO neuron, with two dendrites and one axon (fake geometry).

```python
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=100)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
morpho.R = Cylinder(diameter=1*um, length=150*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs='''
Im = gL * (EL - v) : amp/meter**2
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm, method='exponential_euler')
neuron.v = EL
neuron.I = 0*amp

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron, 'v', record=morpho.R[75*um])

run(1*ms)
neuron.I[morpho.L[50*um]] = 0.2*nA  # injecting in the left dendrite
run(5*ms)
neuron.I = 0*amp
run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[50*um]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[75*um]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for x in linspace(0*um, 100*um, 10, endpoint=False):
    plot(mon_L.t/ms, mon_L[morpho.L[x]].v/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```
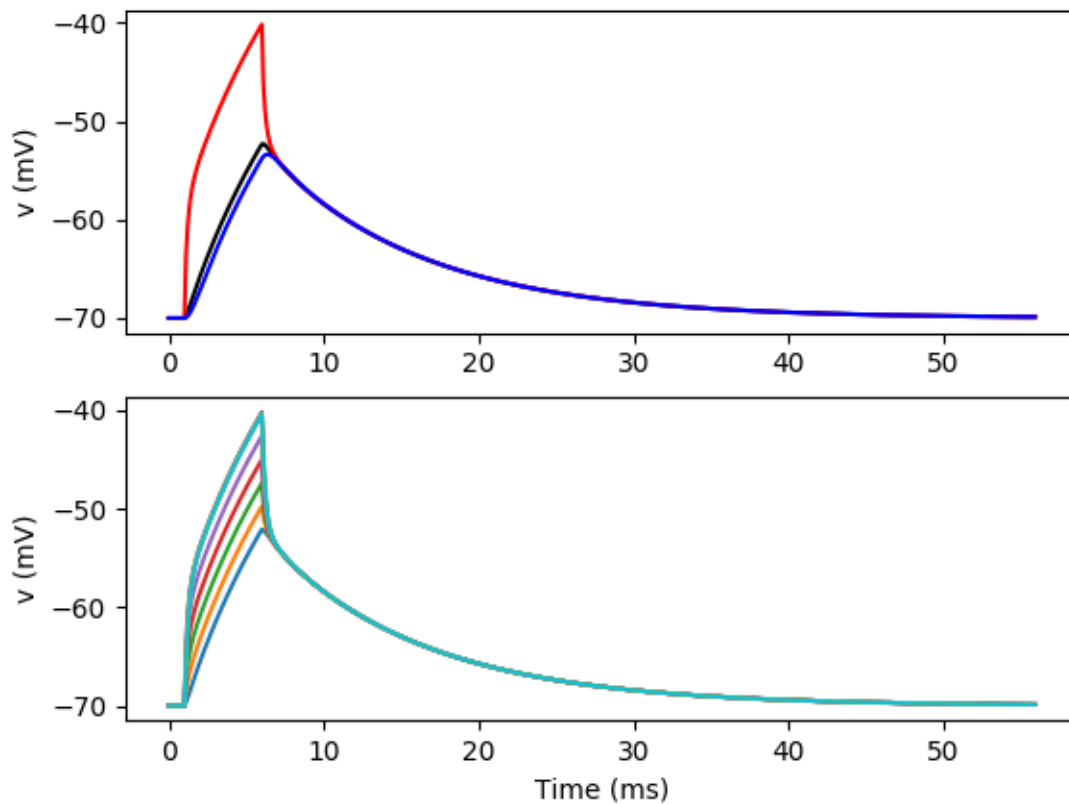
5.10.2 Example: bipolar_with_inputs

A pseudo MSO neuron, with two dendrites (fake geometry). There are synaptic inputs.

```python
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
Es = 0*mV
eqs='''
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
gs : siemens
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
```

```
                            Cm=1*uF/cm**2, Ri=100*ohm*cm, method='exponential_euler')
neuron.v = EL

# Regular inputs
stimulation = NeuronGroup(2, 'dx/dt = 300*Hz : 1', threshold='x>1', reset='x=0',
                          method='euler')
stimulation.x = [0, 0.5]   # Asynchronous

# Synapses
taus = 1*ms
w = 20*nS
S = Synapses(stimulation, neuron, model='''dg/dt = -g/taus : siemens (clock-driven)
                                            gs_post = g : siemens (summed)''',
             on_pre='g += w', method='exact')

S.connect(i=0, j=morpho.L[-1])
S.connect(i=1, j=morpho.R[-1])

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron.R, 'v',
                     record=morpho.R[-1])

run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[-1]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[-1]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for x in linspace(0*um, 100*um, 10, endpoint=False):
    plot(mon_L.t/ms, mon_L[morpho.L[x]].v/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```
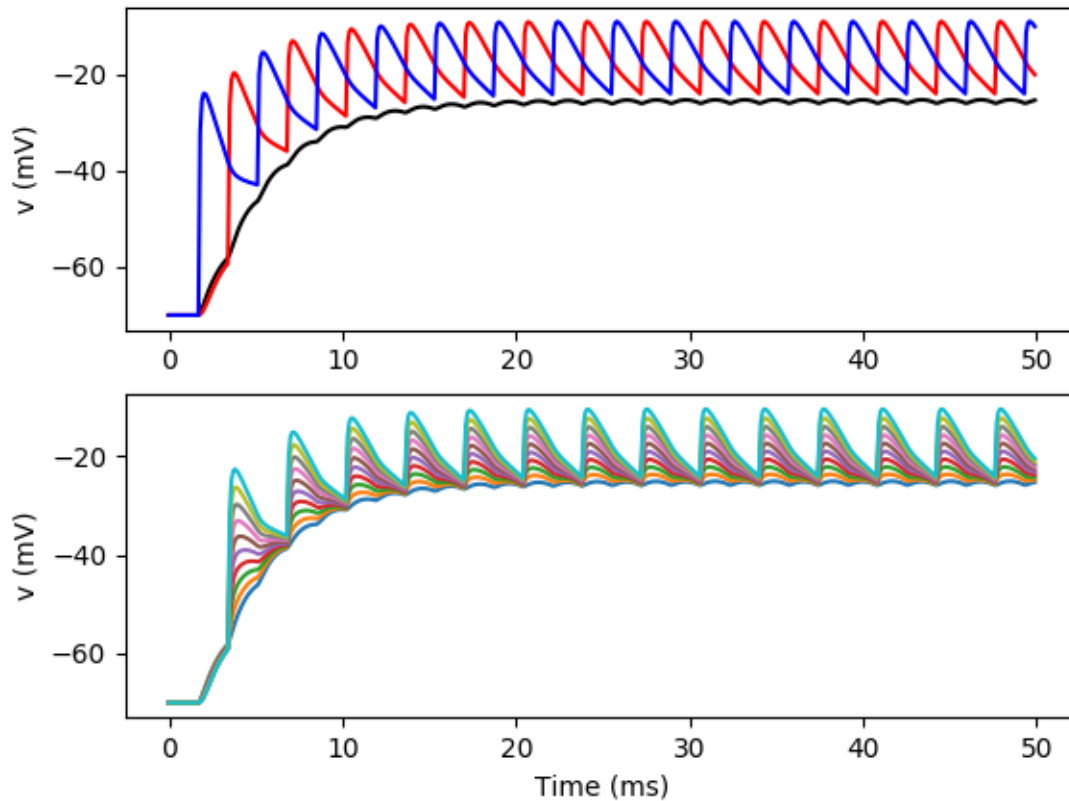
5.10.3 Example: bipolar_with_inputs2

A pseudo MSO neuron, with two dendrites (fake geometry). There are synaptic inputs.

Second method.

```python
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=50)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=50)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
Es = 0*mV
taus = 1*ms
eqs='''
Im = gL*(EL-v) : amp/meter**2
Is = gs*(Es-v) : amp (point current)
dgs/dt = -gs/taus : siemens
'''
```

```
neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm, method='exponential_euler')
neuron.v = EL

# Regular inputs
stimulation = NeuronGroup(2, 'dx/dt = 300*Hz : 1', threshold='x>1', reset='x=0',
                          method='euler')
stimulation.x = [0, 0.5] # Asynchronous

# Synapses
w = 20*nS
S = Synapses(stimulation, neuron, on_pre='gs += w')
S.connect(i=0, j=morpho.L[99.9*um])
S.connect(i=1, j=morpho.R[99.9*um])

# Monitors
mon_soma = StateMonitor(neuron, 'v', record=[0])
mon_L = StateMonitor(neuron.L, 'v', record=True)
mon_R = StateMonitor(neuron, 'v', record=morpho.R[99.9*um])

run(50*ms, report='text')

subplot(211)
plot(mon_L.t/ms, mon_soma[0].v/mV, 'k')
plot(mon_L.t/ms, mon_L[morpho.L[99.9*um]].v/mV, 'r')
plot(mon_L.t/ms, mon_R[morpho.R[99.9*um]].v/mV, 'b')
ylabel('v (mV)')
subplot(212)
for i in [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]:
    plot(mon_L.t/ms, mon_L.v[i, :]/mV)
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```
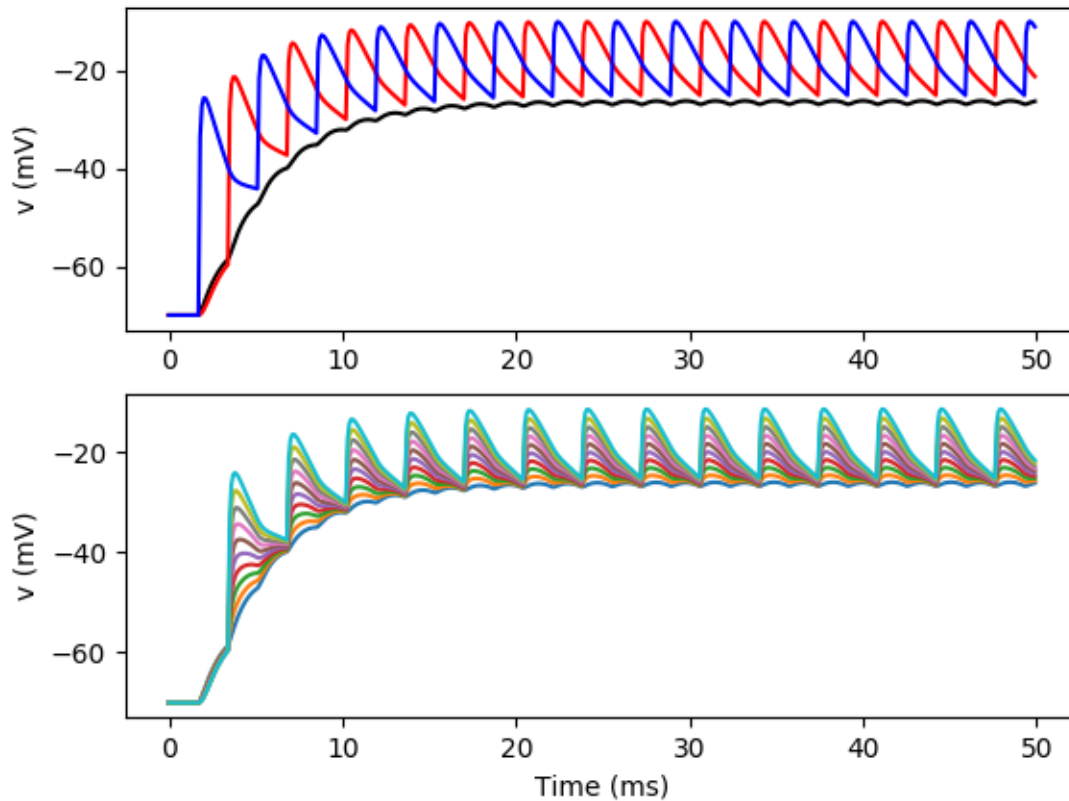
5.10.4 Example: cylinder

A short cylinder with constant injection at one end.

```python
from brian2 import *

defaultclock.dt = 0.01*ms

# Morphology
diameter = 1*um
length = 300*um
Cm = 1*uF/cm**2
Ri = 150*ohm*cm
N = 200
morpho = Cylinder(diameter=diameter, length=length, n=N)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im = gL * (EL - v) : amp/meter**2
I : amp (point current)
```

```
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method='exponential_euler')
neuron.v = EL

la = neuron.space_constant[0]
print("Electrotonic length: %s" % la)

neuron.I[0] = 0.02*nA # injecting at the left end
run(100*ms, report='text')

plot(neuron.distance/um, neuron.v/mV, 'kx')
# Theory
x = neuron.distance
ra = la * 4 * Ri / (pi * diameter**2)
theory = EL + ra * neuron.I[0] * cosh((length - x) / la) / sinh(length / la)
plot(x/um, theory/mV, 'r')
xlabel('x (um)')
ylabel('v (mV)')
show()
```
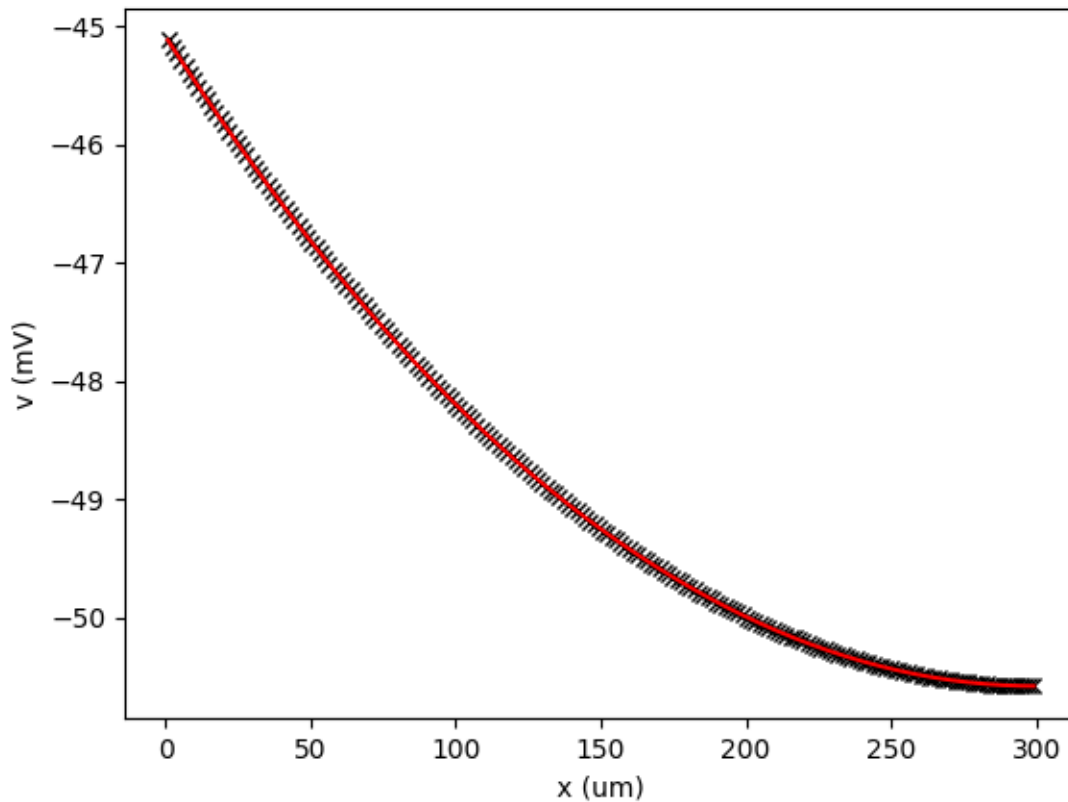
5.10.5 Example: hh_with_spikes

Hodgkin-Huxley equations (1952).

Spikes are recorded along the axon, and then velocity is calculated.

```python
from brian2 import *
from scipy import stats

defaultclock.dt = 0.01*ms

morpho = Cylinder(length=10*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613*mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
I : amp (point current) # applied current
dm/dt = alpham * (1-m) - betam * m : 1
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * 10*mV/exprel((-v+25*mV)/(10*mV))/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * 10*mV/exprel((-v+10*mV)/(10*mV))/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
gNa : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, method="exponential_euler",
                       refractory="m > 0.4", threshold="m > 0.5",
                       Cm=1*uF/cm**2, Ri=35.4*ohm*cm)
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0*amp
neuron.gNa = gNa0
M = StateMonitor(neuron, 'v', record=True)
spikes = SpikeMonitor(neuron)

run(50*ms, report='text')
neuron.I[0] = 1*uA # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(50*ms, report='text')
```

```python
# Calculation of velocity
slope, intercept, r_value, p_value, std_err = stats.linregress(spikes.t/second,
                                                    neuron.distance[spikes.i]/meter)
print("Velocity = %.2f m/s" % slope)

subplot(211)
for i in range(10):
    plot(M.t/ms, M.v.T[:, i*100]/mV)
ylabel('v')
subplot(212)
plot(spikes.t/ms, spikes.i*neuron.length[0]/cm, '.k')
plot(spikes.t/ms, (intercept+slope*(spikes.t/second))/cm, 'r')
xlabel('Time (ms)')
ylabel('Position (cm)')
show()
```
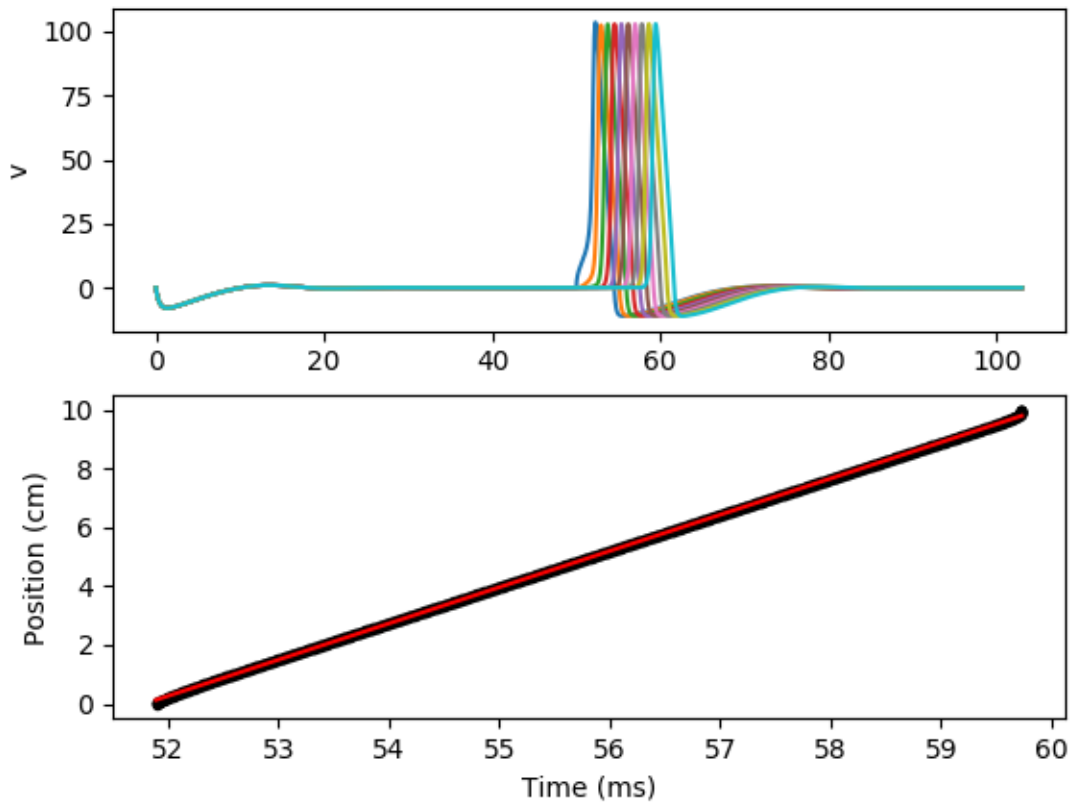
5.10.6 Example: hodgkin_huxley_1952

Hodgkin-Huxley equations (1952).

```python
from brian2 import *

morpho = Cylinder(length=10*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613*mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
I : amp (point current) # applied current
dm/dt = alpham * (1-m) - betam * m : 1
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * 10*mV/exprel((-v+25*mV)/(10*mV))/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * 10*mV/exprel((-v+10*mV)/(10*mV))/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
gNa : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2,
                       Ri=35.4*ohm*cm, method="exponential_euler")
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0
neuron.gNa = gNa0
neuron[5*cm:10*cm].gNa = 0*siemens/cm**2
M = StateMonitor(neuron, 'v', record=True)

run(50*ms, report='text')
neuron.I[0] = 1*uA  # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(100*ms, report='text')
for i in range(75, 125, 1):
    plot(cumsum(neuron.length)/cm, i+(1./60)*M.v[:, i*5]/mV, 'k')
yticks([])
ylabel('Time [major] v (mV) [minor]')
xlabel('Position (cm)')
axis('tight')
show()
```
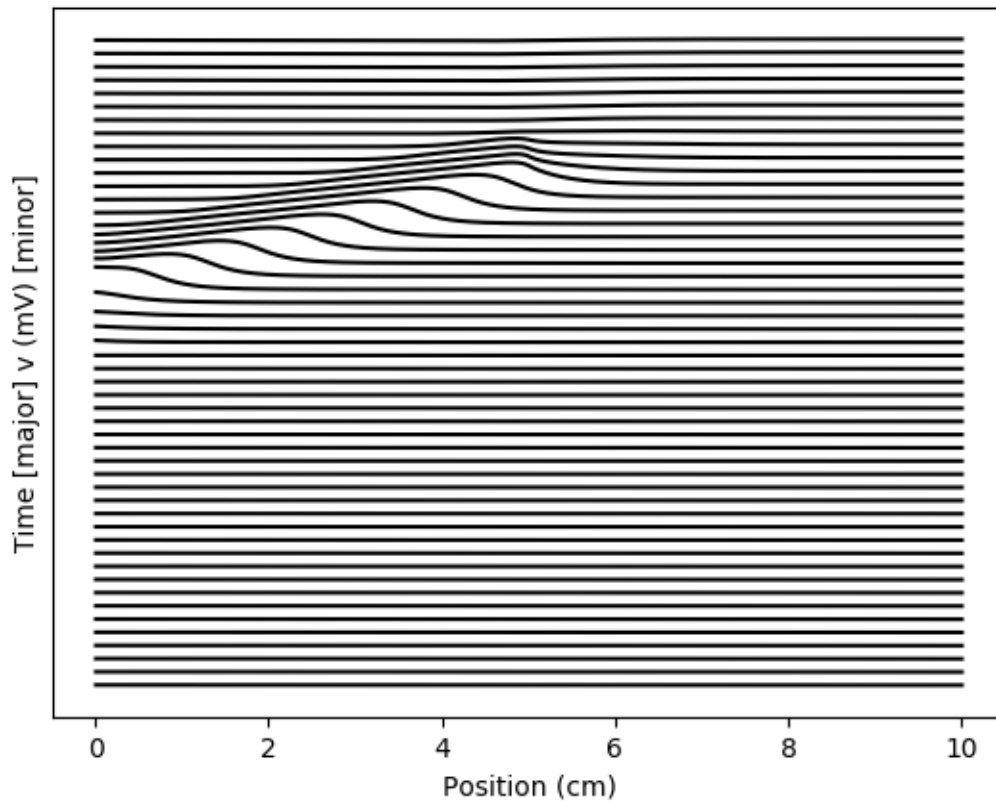
5.10.7 Example: infinite_cable

An (almost) infinite cable with pulse injection in the middle.

```python
from brian2 import *

defaultclock.dt = 0.001*ms

# Morphology
diameter = 1*um
Cm = 1*uF/cm**2
Ri = 100*ohm*cm
N = 500
morpho = Cylinder(diameter=diameter, length=3*mm, n=N)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im = gL * (EL-v) : amp/meter**2
I : amp (point current)
'''
```

```python
neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method = 'exponential_euler')
neuron.v = EL

taum = Cm  /gL  # membrane time constant
print("Time constant: %s" % taum)
la = neuron.space_constant[0]
print("Characteristic length: %s" % la)

# Monitors
mon = StateMonitor(neuron, 'v', record=range(0, N//2, 20))

neuron.I[len(neuron) // 2] = 1*nA  # injecting in the middle
run(0.02*ms)
neuron.I = 0*amp
run(10*ms, report='text')

t = mon.t
plot(t/ms, mon.v.T/mV, 'k')
# Theory (incorrect near cable ends)
for i in range(0, len(neuron)//2, 20):
    x = (len(neuron)/2 - i) * morpho.length[0]
    theory = (1/(la*Cm*pi*diameter) * sqrt(taum / (4*pi*(t + defaultclock.dt))) *
              exp(-(t+defaultclock.dt)/taum -
                  taum / (4*(t+defaultclock.dt))*(x/la)**2))
    theory = EL + theory * 1*nA * 0.02*ms
    plot(t/ms, theory/mV, 'r')
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```
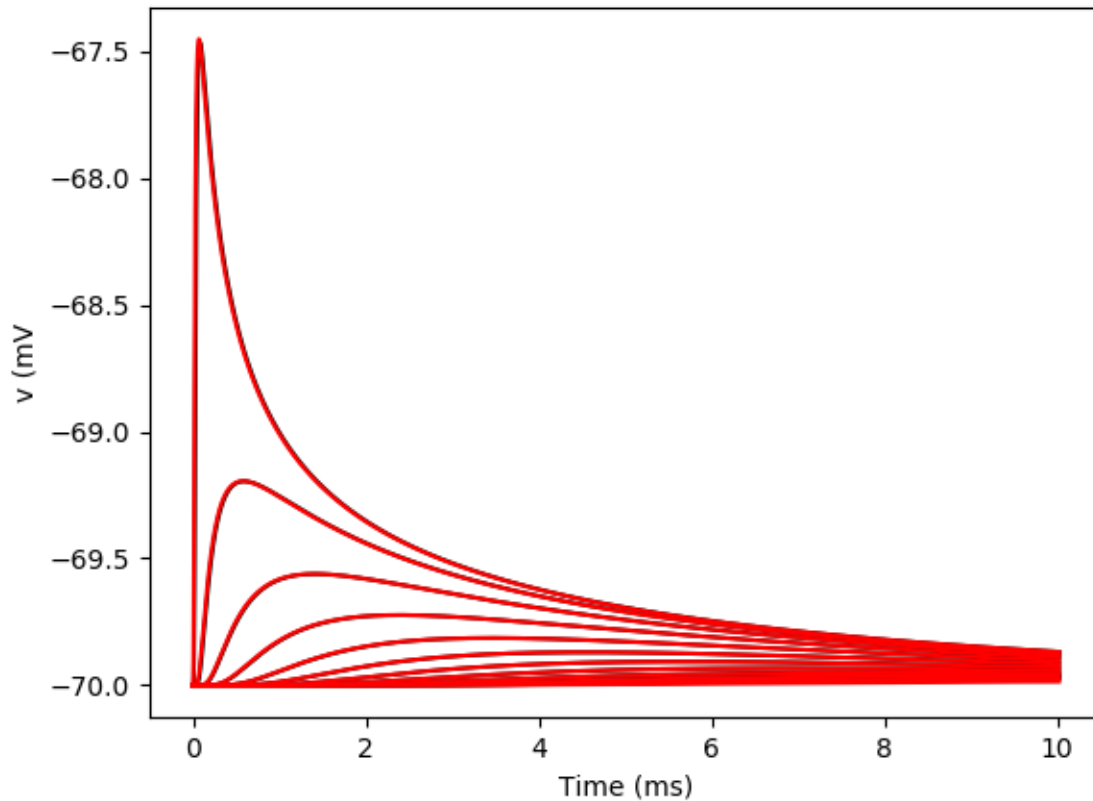
5.10.8 Example: lfp

Hodgkin-Huxley equations (1952)

We calculate the extracellular field potential at various places.

```python
from brian2 import *
defaultclock.dt = 0.01*ms
morpho = Cylinder(x=[0, 10]*cm, diameter=2*238*um, n=1000, type='axon')

El = 10.613* mV
ENa = 115*mV
EK = -12*mV
gl = 0.3*msiemens/cm**2
gNa0 = 120*msiemens/cm**2
gK = 36*msiemens/cm**2

# Typical equations
eqs = '''
# The same equations for the whole neuron, but possibly different parameter values
# distributed transmembrane current
Im = gl * (El-v) + gNa * m**3 * h * (ENa-v) + gK * n**4 * (EK-v) : amp/meter**2
I : amp (point current) # applied current
```

(continues on next page)

```
dm/dt = alpham * (1-m) - betam * m : 1
dn/dt = alphan * (1-n) - betan * n : 1
dh/dt = alphah * (1-h) - betah * h : 1
alpham = (0.1/mV) * 10*mV/exprel((-v+25*mV)/(10*mV))/ms : Hz
betam = 4 * exp(-v/(18*mV))/ms : Hz
alphah = 0.07 * exp(-v/(20*mV))/ms : Hz
betah = 1/(exp((-v+30*mV) / (10*mV)) + 1)/ms : Hz
alphan = (0.01/mV) * 10*mV/exprel((-v+10*mV)/(10*mV))/ms : Hz
betan = 0.125*exp(-v/(80*mV))/ms : Hz
gNa : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=1*uF/cm**2,
                       Ri=35.4*ohm*cm, method="exponential_euler")
neuron.v = 0*mV
neuron.h = 1
neuron.m = 0
neuron.n = .5
neuron.I = 0
neuron.gNa = gNa0
neuron[5*cm:10*cm].gNa = 0*siemens/cm**2
M = StateMonitor(neuron, 'v', record=True)

# LFP recorder
Ne = 5 # Number of electrodes
sigma = 0.3*siemens/meter # Resistivity of extracellular field (0.3-0.4 S/m)
lfp = NeuronGroup(Ne, model='''v : volt
                               x : meter
                               y : meter
                               z : meter''')
lfp.x = 7*cm # Off center (to be far from stimulating electrode)
lfp.y = [1*mm, 2*mm, 4*mm, 8*mm, 16*mm]
S = Synapses(neuron, lfp, model='''w : ohm*meter**2 (constant) # Weight in the LFP␣
↪calculation
                                   v_post = w*(Ic_pre-Im_pre) : volt (summed)''')
S.summed_updaters['v_post'].when = 'after_groups'  # otherwise Ic has not yet been␣
↪updated for the current time step.
S.connect()
S.w = 'area_pre/(4*pi*sigma)/((x_pre-x_post)**2+(y_pre-y_post)**2+(z_pre-z_
↪post)**2)**.5'

Mlfp = StateMonitor(lfp, 'v', record=True)

run(50*ms, report='text')
neuron.I[0] = 1*uA  # current injection at one end
run(3*ms)
neuron.I = 0*amp
run(100*ms, report='text')

subplot(211)
for i in range(10):
    plot(M.t/ms, M.v[i*100]/mV)
ylabel('$V_m$ (mV)')
subplot(212)
for i in range(5):
    plot(M.t/ms, Mlfp.v[i]/mV)
ylabel('LFP (mV)')
```

```
xlabel('Time (ms)')
show()
```
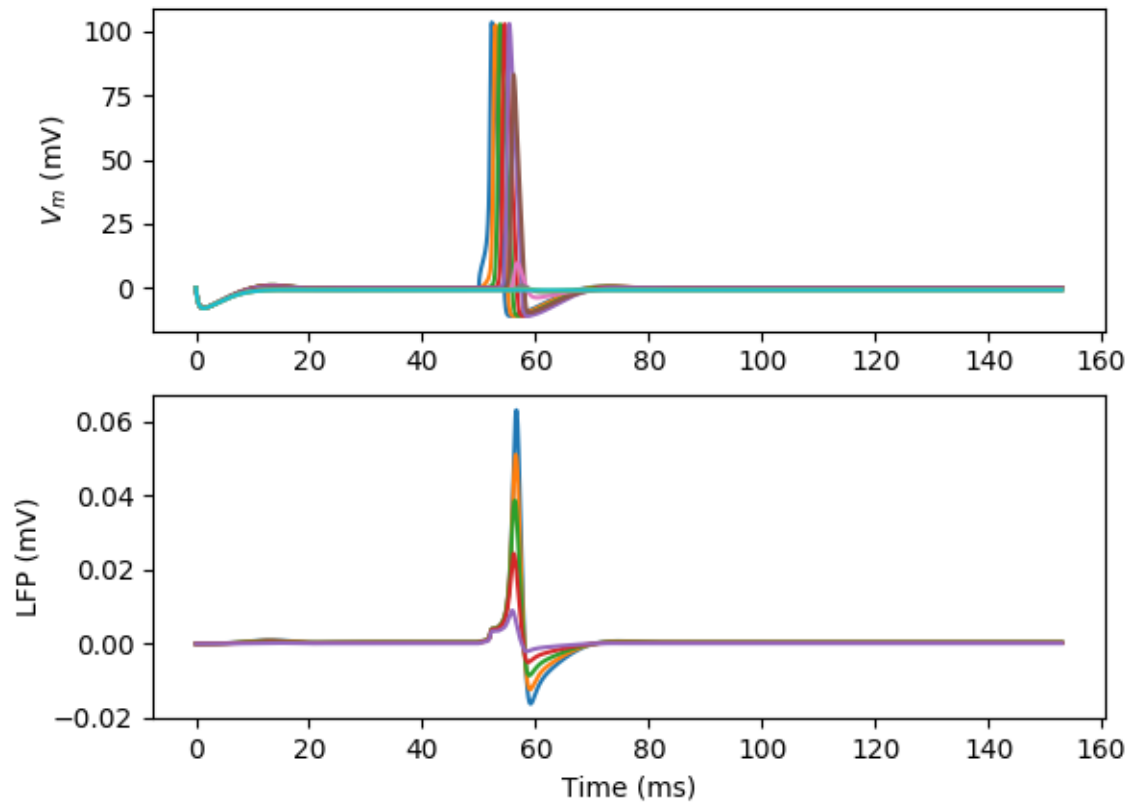

5.10.9 Example: morphotest

Demonstrate the usage of the `Morphology` object.

```python
from brian2 import *

# Morphology
morpho = Soma(30*um)
morpho.L = Cylinder(diameter=1*um, length=100*um, n=5)
morpho.LL = Cylinder(diameter=1*um, length=20*um, n=2)
morpho.R = Cylinder(diameter=1*um, length=100*um, n=5)

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
eqs = '''
Im = gL * (EL-v) : amp/meter**2
'''
```

```
neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm, method='exponential_euler')
neuron.v = arange(0, 13)*volt

print(neuron.v)
print(neuron.L.v)
print(neuron.LL.v)
print(neuron.L.main.v)
```

5.10.10 Example: rall

A cylinder plus two branches, with diameters according to Rall's formula

```
from brian2 import *

defaultclock.dt = 0.01*ms

# Passive channels
gL = 1e-4*siemens/cm**2
EL = -70*mV

# Morphology
diameter = 1*um
length = 300*um
Cm = 1*uF/cm**2
Ri = 150*ohm*cm
N = 500
rm = 1 / (gL * pi * diameter)  # membrane resistance per unit length
ra = (4 * Ri)/(pi * diameter**2)   # axial resistance per unit length
la = sqrt(rm / ra) # space length
morpho = Cylinder(diameter=diameter, length=length, n=N)
d1 = 0.5*um
L1 = 200*um
rm = 1 / (gL * pi * d1) # membrane resistance per unit length
ra = (4 * Ri) / (pi * d1**2) # axial resistance per unit length
l1 = sqrt(rm / ra) # space length
morpho.L = Cylinder(diameter=d1, length=L1, n=N)
d2 = (diameter**1.5 - d1**1.5)**(1. / 1.5)
rm = 1/(gL * pi * d2) # membrane resistance per unit length
ra = (4 * Ri) / (pi * d2**2) # axial resistance per unit length
l2 = sqrt(rm / ra) # space length
L2 = (L1 / l1) * l2
morpho.R = Cylinder(diameter=d2, length=L2, n=N)

eqs='''
Im = gL * (EL-v) : amp/meter**2
I : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method='exponential_euler')
neuron.v = EL
```

```
neuron.I[0] = 0.02*nA # injecting at the left end
run(100*ms, report='text')

plot(neuron.main.distance/um, neuron.main.v/mV, 'k')
plot(neuron.L.distance/um, neuron.L.v/mV, 'k')
plot(neuron.R.distance/um, neuron.R.v/mV, 'k')
# Theory
x = neuron.main.distance
ra = la * 4 * Ri/(pi * diameter**2)
l = length/la + L1/l1
theory = EL + ra*neuron.I[0]*cosh(l - x/la)/sinh(l)
plot(x/um, theory/mV, 'r')
x = neuron.L.distance
theory = (EL+ra*neuron.I[0]*cosh(l - neuron.main.distance[-1]/la -
                                 (x - neuron.main.distance[-1])/l1)/sinh(l))
plot(x/um, theory/mV, 'r')
x = neuron.R.distance
theory = (EL+ra*neuron.I[0]*cosh(l - neuron.main.distance[-1]/la -
                                 (x - neuron.main.distance[-1])/l2)/sinh(l))
plot(x/um, theory/mV, 'r')
xlabel('x (um)')
ylabel('v (mV)')
show()
```
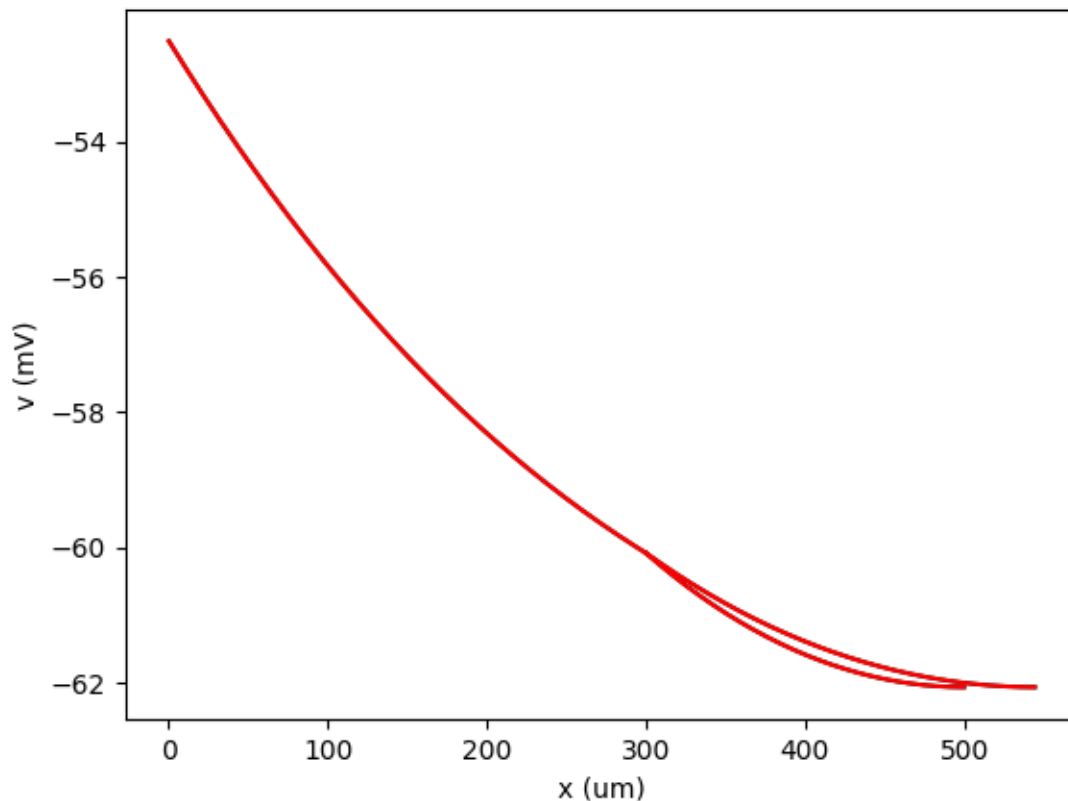
5.10.11 Example: spike_initiation

Ball and stick with Na and K channels

```python
from brian2 import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(30*um)
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=100)

# Channels
gL = 1e-4*siemens/cm**2
EL = -70*mV
ENa = 50*mV
ka = 6*mV
ki = 6*mV
va = -30*mV
vi = -50*mV
EK = -90*mV
vk = -20*mV
kk = 8*mV
eqs = '''
Im = gL*(EL-v)+gNa*m*h*(ENa-v)+gK*n*(EK-v) : amp/meter**2
dm/dt = (minf-m)/(0.3*ms) : 1 # simplified Na channel
dh/dt = (hinf-h)/(3*ms) : 1 # inactivation
dn/dt = (ninf-n)/(5*ms) : 1 # K+
minf = 1/(1+exp((va-v)/ka)) : 1
hinf = 1/(1+exp((v-vi)/ki)) : 1
ninf = 1/(1+exp((vk-v)/kk)) : 1
I : amp (point current)
gNa : siemens/meter**2
gK : siemens/meter**2
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs,
                       Cm=1*uF/cm**2, Ri=100*ohm*cm, method='exponential_euler')
neuron.v = -65*mV
neuron.I = 0*amp
neuron.axon[30*um:60*um].gNa = 700*gL
neuron.axon[30*um:60*um].gK = 700*gL

# Monitors
mon=StateMonitor(neuron, 'v', record=True)

run(1*ms)
neuron.main.I = 0.15*nA
run(50*ms)
neuron.I = 0*amp
run(95*ms, report='text')

plot(mon.t/ms, mon.v[0]/mV, 'r')
plot(mon.t/ms, mon.v[20]/mV, 'g')
plot(mon.t/ms, mon.v[40]/mV, 'b')
plot(mon.t/ms, mon.v[60]/mV, 'k')
```

```
plot(mon.t/ms, mon.v[80]/mV, 'y')
xlabel('Time (ms)')
ylabel('v (mV)')
show()
```
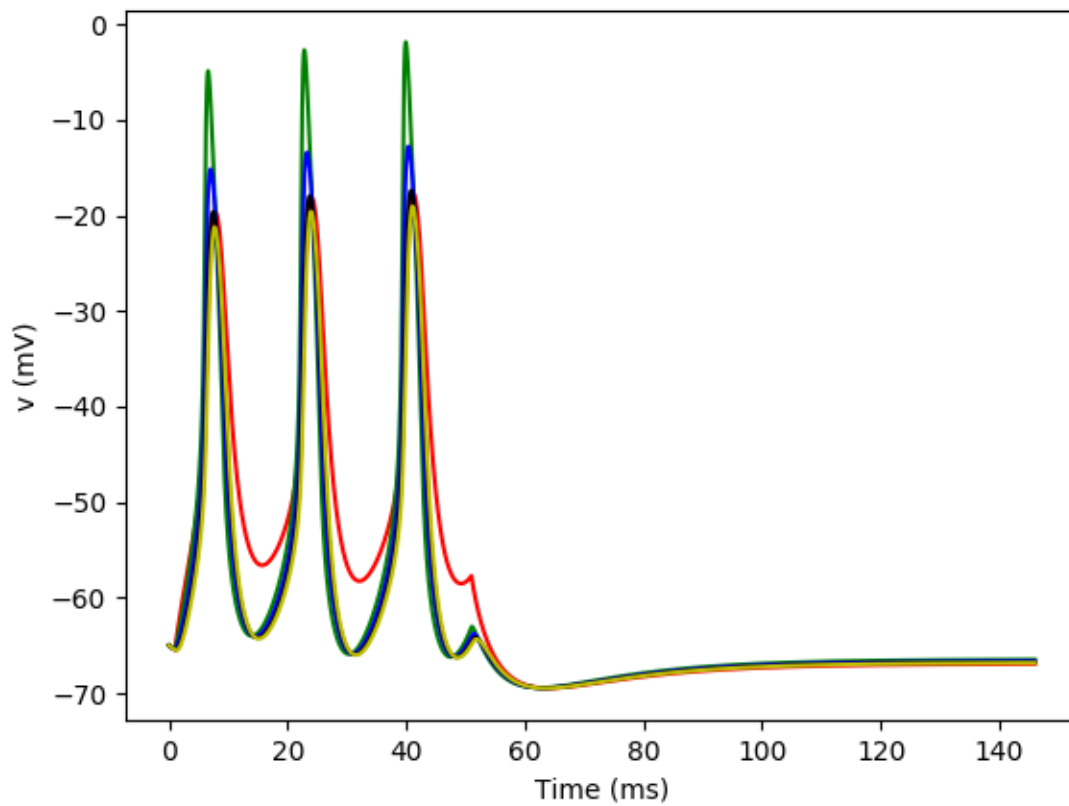

5.11 frompapers

5.11.1 Example: Brette_2004

Phase locking in leaky integrate-and-fire model

Fig. 2A from:

Brette R (2004). Dynamics of one-dimensional spiking neuron models. J Math Biol 48(1): 38-56.

This shows the phase-locking structure of a LIF driven by a sinusoidal current. When the current crosses the threshold (a<3), the model almost always phase locks (in a measure-theoretical sense).

```python
from brian2 import *

# defaultclock.dt = 0.01*ms  # for a more precise picture
N = 2000
tau = 100*ms
freq = 1/tau

eqs = '''
dv/dt = (-v + a + 2*sin(2*pi*t/tau))/tau : 1
a : 1
'''

neurons = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', method='euler')
neurons.a = linspace(2, 4, N)

run(5*second, report='text')  # discard the first spikes (wait for convergence)
S = SpikeMonitor(neurons)
run(5*second, report='text')

i, t = S.it
plot((t % tau)/tau, neurons.a[i], ',')
xlabel('Spike phase')
ylabel('Parameter a')
show()
```
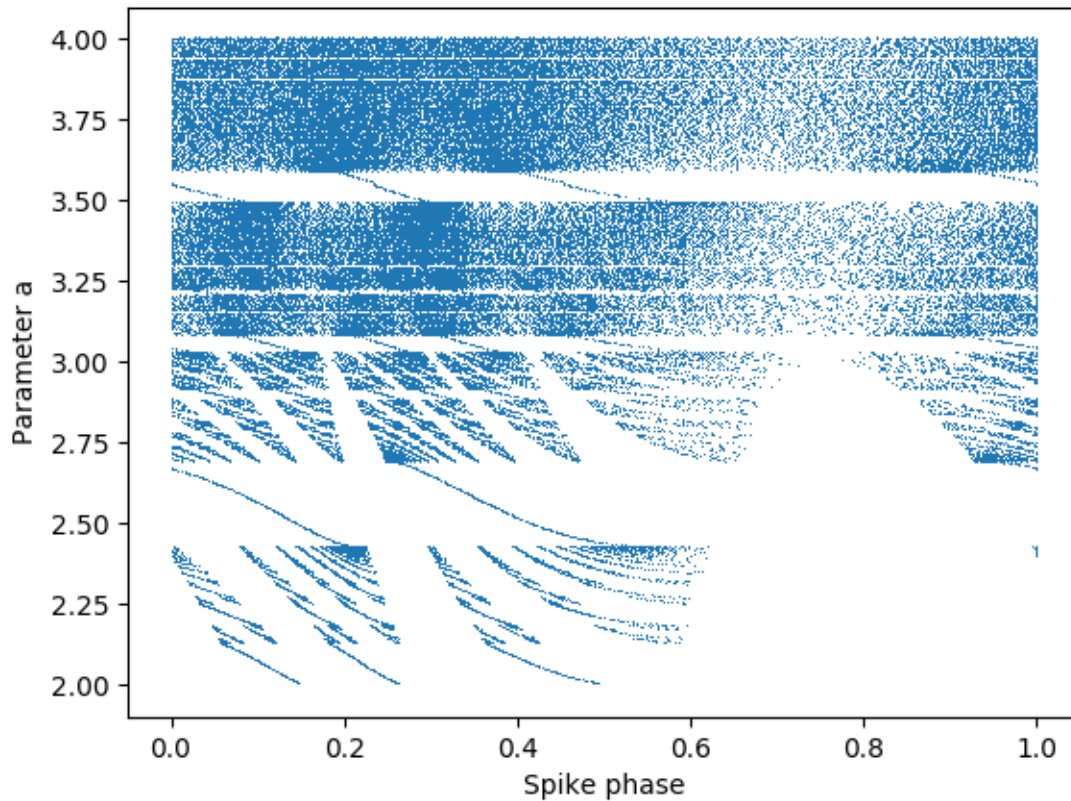
5.11.2 Example: Brette_Gerstner_2005

Adaptive exponential integrate-and-fire model.

http://www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model

Introduced in Brette R. and Gerstner W. (2005), Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity, J. Neurophysiol. 94: 3637 - 3642.

```python
from brian2 import *

# Parameters
C = 281 * pF
gL = 30 * nS
taum = C / gL
EL = -70.6 * mV
VT = -50.4 * mV
DeltaT = 2 * mV
Vcut = VT + 5 * DeltaT

# Pick an electrophysiological behaviour
tauw, a, b, Vr = 144*ms, 4*nS, 0.0805*nA, -70.6*mV # Regular spiking (as in the paper)
#tauw,a,b,Vr=20*ms,4*nS,0.5*nA,VT+5*mV # Bursting
```

```python
#tauw,a,b,Vr=144*ms,2*C/(144*ms),0*nA,-70.6*mV # Fast spiking

eqs = """
dvm/dt = (gL*(EL - vm) + gL*DeltaT*exp((vm - VT)/DeltaT) + I - w)/C : volt
dw/dt = (a*(vm - EL) - w)/tauw : amp
I : amp
"""

neuron = NeuronGroup(1, model=eqs, threshold='vm>Vcut',
                     reset="vm=Vr; w+=b", method='euler')
neuron.vm = EL
trace = StateMonitor(neuron, 'vm', record=0)
spikes = SpikeMonitor(neuron)

run(20 * ms)
neuron.I = 1*nA
run(100 * ms)
neuron.I = 0*nA
run(20 * ms)

# We draw nicer spikes
vm = trace[0].vm[:]
for t in spikes.t:
    i = int(t / defaultclock.dt)
    vm[i] = 20*mV

plot(trace.t / ms, vm / mV)
xlabel('time (ms)')
ylabel('membrane potential (mV)')
show()
```
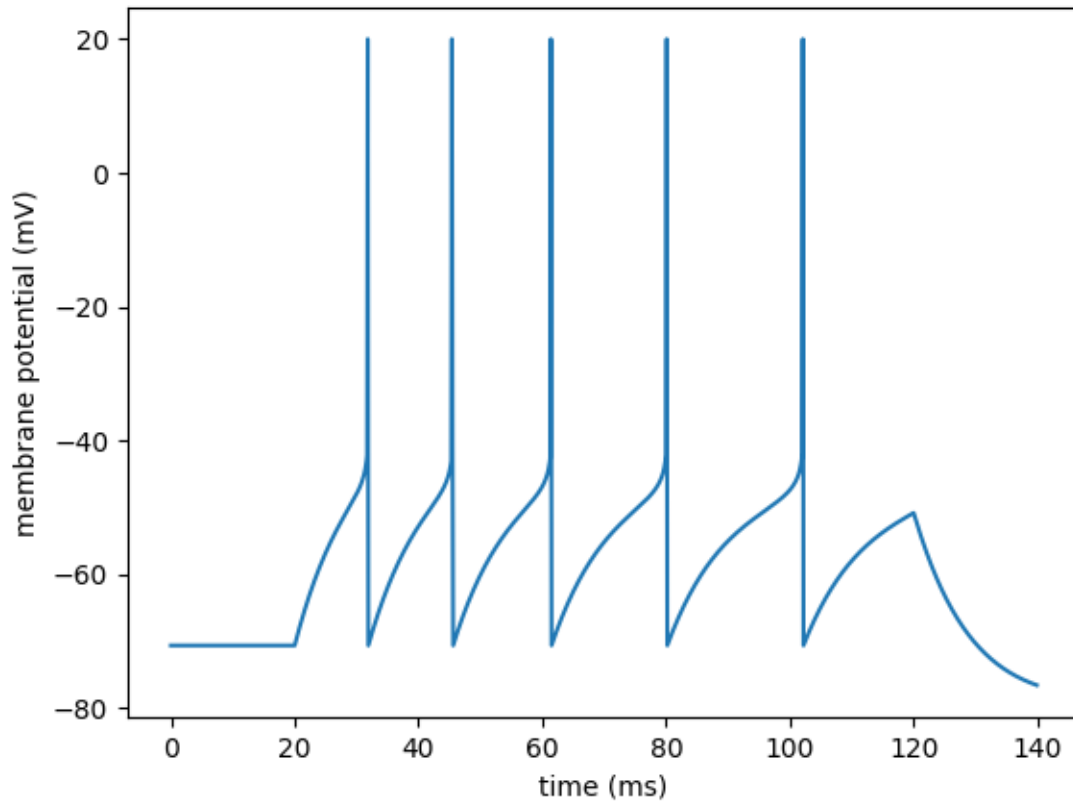
5.11.3 Example: Brette_Guigon_2003

Reliability of spike timing

Adapted from Fig. 10D,E of

Brette R and E Guigon (2003). Reliability of Spike Timing Is a General Property of Spiking Model Neurons. Neural Computation 15, 279-308.

This shows that reliability of spike timing is a generic property of spiking neurons, even those that are not leaky. This is a non-physiological model which can be leaky or anti-leaky depending on the sign of the input I.

All neurons receive the same fluctuating input, scaled by a parameter p that varies across neurons. This shows:

1. reproducibility of spike timing

2. robustness with respect to deterministic changes (parameter)

3. increased reproducibility in the fluctuation-driven regime (input crosses the threshold)

```
from brian2 import *

N = 500
```

(continues on next page)

```python
tau = 33*ms
taux = 20*ms
sigma = 0.02

eqs_input = '''
dx/dt = -x/taux + (2/taux)**.5*xi : 1
'''

eqs = '''
dv/dt = (v*I + 1)/tau + sigma*(2/tau)**.5*xi : 1
I = 0.5 + 3*p*B : 1
B = 2./(1 + exp(-2*x)) - 1 : 1 (shared)
p : 1
x : 1 (linked)
'''

input = NeuronGroup(1, eqs_input, method='euler')
neurons = NeuronGroup(N, eqs, threshold='v>1', reset='v=0', method='euler')
neurons.p = '1.0*i/N'
neurons.v = 'rand()'
neurons.x = linked_var(input, 'x')

M = StateMonitor(neurons, 'B', record=0)
S = SpikeMonitor(neurons)

run(1000*ms, report='text')

subplot(211)  # The input
plot(M.t/ms, M[0].B)
xticks([])
title('shared input')
subplot(212)
plot(S.t/ms, neurons.p[S.i], ',')
plot([0, 1000], [.5, .5], color='C1')
xlabel('time (ms)')
ylabel('p')
title('spiking activity')
show()
```
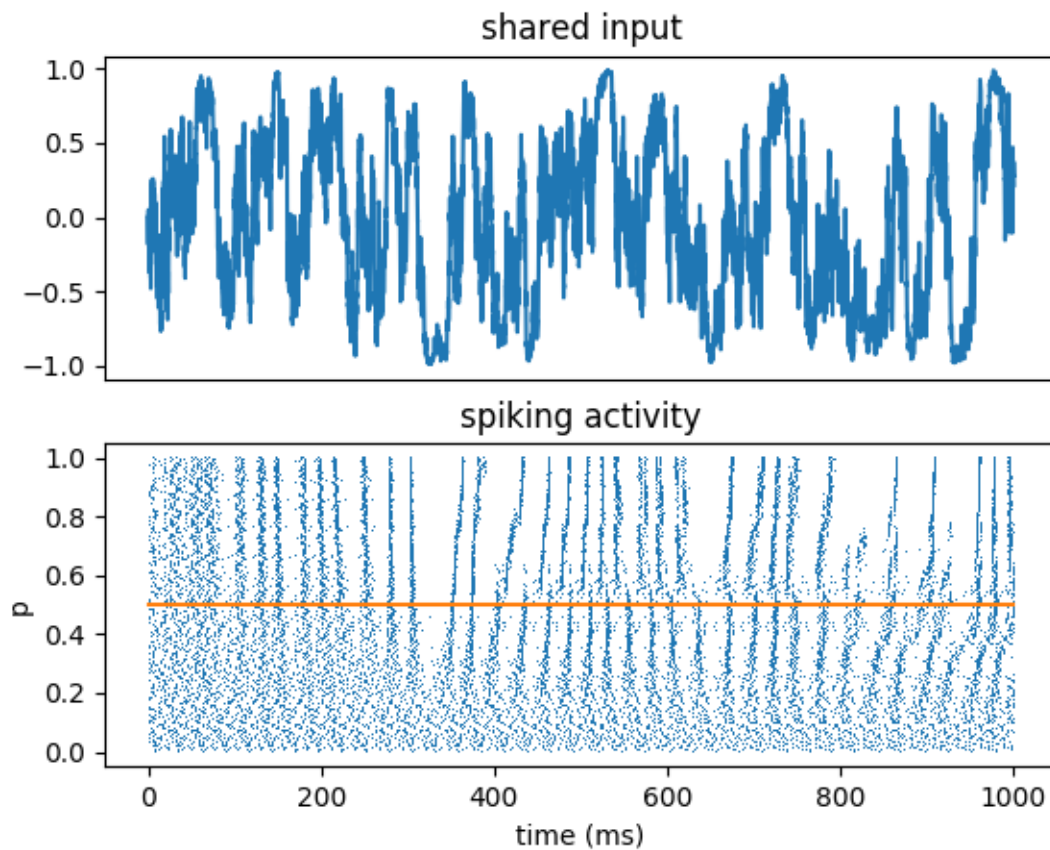
5.11.4 Example: Brunel_Hakim_1999

Dynamics of a network of sparsely connected inhibitory current-based integrate-and-fire neurons. Individual neurons fire irregularly at low rate but the network is in an oscillatory global activity regime where neurons are weakly synchronized.

Reference: "Fast Global Oscillations in Networks of Integrate-and-Fire Neurons with Low Firing Rates" Nicolas Brunel & Vincent Hakim Neural Computation 11, 1621-1671 (1999)

```
from brian2 import *

N = 5000
Vr = 10*mV
theta = 20*mV
tau = 20*ms
delta = 2*ms
taurefr = 2*ms
duration = .1*second
C = 1000
sparseness = float(C)/N
J = .1*mV
muext = 25*mV
sigmaext = 1*mV
```

```
eqs = """
dV/dt = (-V+muext + sigmaext * sqrt(tau) * xi)/tau : volt
"""

group = NeuronGroup(N, eqs, threshold='V>theta',
                    reset='V=Vr', refractory=taurefr, method='euler')
group.V = Vr
conn = Synapses(group, group, on_pre='V += -J', delay=delta)
conn.connect(p=sparseness)
M = SpikeMonitor(group)
LFP = PopulationRateMonitor(group)

run(duration)

subplot(211)
plot(M.t/ms, M.i, '.')
xlim(0, duration/ms)

subplot(212)
plot(LFP.t/ms, LFP.smooth_rate(window='flat', width=0.5*ms)/Hz)
xlim(0, duration/ms)

show()
```
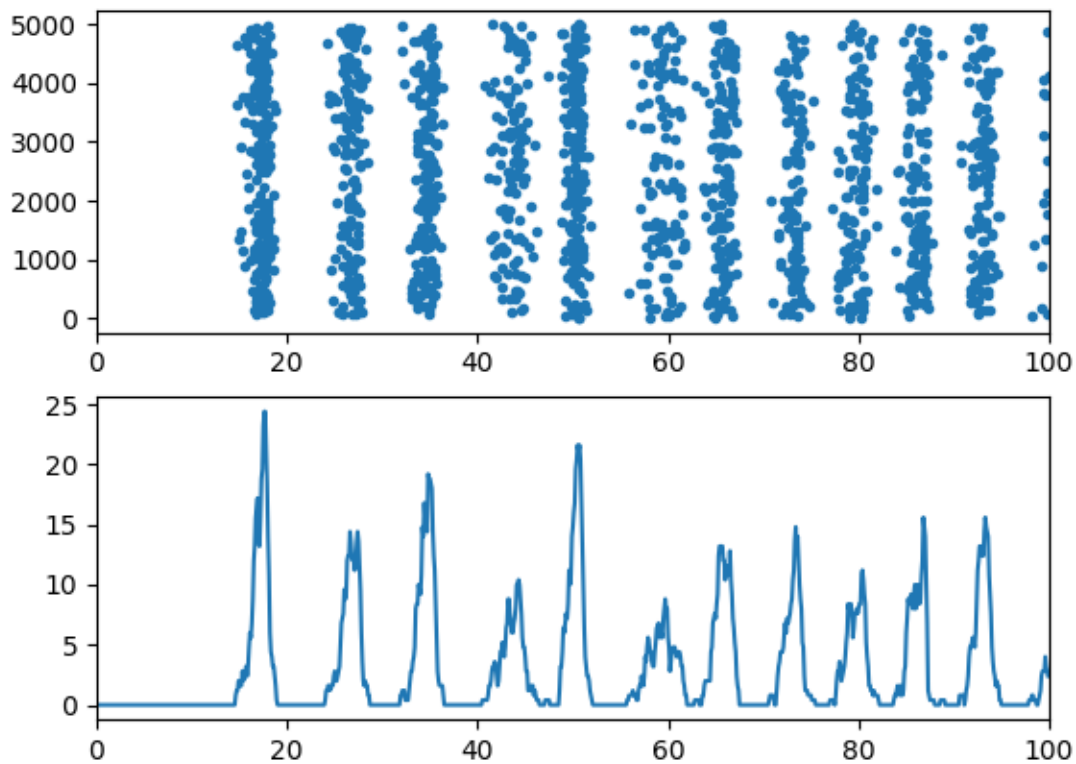
5.11.5 Example: Brunel_Wang_2001

Sample-specific persistent activity

Five subpopulations with three selective and one reset stimuli example. Analog to figure 6b in the paper.

BRUNEL, Nicolas et WANG, Xiao-Jing. Effects of neuromodulation in a cortical network model of object working memory dominated by recurrent inhibition. Journal of computational neuroscience, 2001, vol. 11, no 1, p. 63-85.

```python
from brian2 import *

# populations
N = 1000
N_E = int(N * 0.8)  # pyramidal neurons
N_I = int(N * 0.2)  # interneurons

# voltage
V_L = -70. * mV
V_thr = -50. * mV
V_reset = -55. * mV
V_E = 0. * mV
V_I = -70. * mV

# membrane capacitance
C_m_E = 0.5 * nF
C_m_I = 0.2 * nF

# membrane leak
g_m_E = 25. * nS
g_m_I = 20. * nS

# refractory period
tau_rp_E = 2. * ms
tau_rp_I = 1. * ms

# external stimuli
rate = 3 * Hz
C_ext = 800

# synapses
C_E = N_E
C_I = N_I

# AMPA (excitatory)
g_AMPA_ext_E = 2.08 * nS
g_AMPA_rec_E = 0.104 * nS * 800. / N_E
g_AMPA_ext_I = 1.62 * nS
g_AMPA_rec_I = 0.081 * nS * 800. / N_E
tau_AMPA = 2. * ms

# NMDA (excitatory)
g_NMDA_E = 0.327 * nS * 800. / N_E
g_NMDA_I = 0.258 * nS * 800. / N_E
tau_NMDA_rise = 2. * ms
tau_NMDA_decay = 100. * ms
```

```
alpha = 0.5 / ms
Mg2 = 1.

# GABAergic (inhibitory)
g_GABA_E = 1.25 * nS * 200. / N_I
g_GABA_I = 0.973 * nS * 200. / N_I
tau_GABA = 10. * ms

# subpopulations
f = 0.1
p = 5
N_sub = int(N_E * f)
N_non = int(N_E * (1. - f * p))
w_plus = 2.1
w_minus = 1. - f * (w_plus - 1.) / (1. - f)

# modeling
eqs_E = '''
dv / dt = (- g_m_E * (v - V_L) - I_syn) / C_m_E : volt (unless refractory)

I_syn = I_AMPA_ext + I_AMPA_rec + I_NMDA_rec + I_GABA_rec : amp

I_AMPA_ext = g_AMPA_ext_E * (v - V_E) * s_AMPA_ext : amp
I_AMPA_rec = g_AMPA_rec_E * (v - V_E) * 1 * s_AMPA : amp
ds_AMPA_ext / dt = - s_AMPA_ext / tau_AMPA : 1
ds_AMPA / dt = - s_AMPA / tau_AMPA : 1

I_NMDA_rec = g_NMDA_E * (v - V_E) / (1 + Mg2 * exp(-0.062 * v / mV) / 3.57) * s_NMDA_
→tot : amp
s_NMDA_tot : 1

I_GABA_rec = g_GABA_E * (v - V_I) * s_GABA : amp
ds_GABA / dt = - s_GABA / tau_GABA : 1
'''

eqs_I = '''
dv / dt = (- g_m_I * (v - V_L) - I_syn) / C_m_I : volt (unless refractory)

I_syn = I_AMPA_ext + I_AMPA_rec + I_NMDA_rec + I_GABA_rec : amp

I_AMPA_ext = g_AMPA_ext_I * (v - V_E) * s_AMPA_ext : amp
I_AMPA_rec = g_AMPA_rec_I * (v - V_E) * 1 * s_AMPA : amp
ds_AMPA_ext / dt = - s_AMPA_ext / tau_AMPA : 1
ds_AMPA / dt = - s_AMPA / tau_AMPA : 1

I_NMDA_rec = g_NMDA_I * (v - V_E) / (1 + Mg2 * exp(-0.062 * v / mV) / 3.57) * s_NMDA_
→tot : amp
s_NMDA_tot : 1

I_GABA_rec = g_GABA_I * (v - V_I) * s_GABA : amp
ds_GABA / dt = - s_GABA / tau_GABA : 1
'''

P_E = NeuronGroup(N_E, eqs_E, threshold='v > V_thr', reset='v = V_reset',␣
→refractory=tau_rp_E, method='euler')
P_E.v = V_L
P_I = NeuronGroup(N_I, eqs_I, threshold='v > V_thr', reset='v = V_reset',␣
→refractory=tau_rp_I, method='euler')
```

```python
P_I.v = V_L


eqs_glut = '''
s_NMDA_tot_post = w * s_NMDA : 1 (summed)
ds_NMDA / dt = - s_NMDA / tau_NMDA_decay + alpha * x * (1 - s_NMDA) : 1 (clock-driven)
dx / dt = - x / tau_NMDA_rise : 1 (clock-driven)
w : 1
'''


eqs_pre_glut = '''
s_AMPA += w
x += 1
'''


eqs_pre_gaba = '''
s_GABA += 1
'''


eqs_pre_ext = '''
s_AMPA_ext += 1
'''


# E to E
C_E_E = Synapses(P_E, P_E, model=eqs_glut, on_pre=eqs_pre_glut, method='euler')
C_E_E.connect('i != j')
C_E_E.w[:] = 1

for pi in range(N_non, N_non + p * N_sub, N_sub):

    # internal other subpopulation to current nonselective
    C_E_E.w[C_E_E.indices[:, pi:pi + N_sub]] = w_minus

    # internal current subpopulation to current subpopulation
    C_E_E.w[C_E_E.indices[pi:pi + N_sub, pi:pi + N_sub]] = w_plus

# E to I
C_E_I = Synapses(P_E, P_I, model=eqs_glut, on_pre=eqs_pre_glut, method='euler')
C_E_I.connect()
C_E_I.w[:] = 1

# I to I
C_I_I = Synapses(P_I, P_I, on_pre=eqs_pre_gaba, method='euler')
C_I_I.connect('i != j')

# I to E
C_I_E = Synapses(P_I, P_E, on_pre=eqs_pre_gaba, method='euler')
C_I_E.connect()

# external noise
C_P_E = PoissonInput(P_E, 's_AMPA_ext', C_ext, rate, '1')
C_P_I = PoissonInput(P_I, 's_AMPA_ext', C_ext, rate, '1')

# at 1s, select population 1
C_selection = int(f * C_ext)
rate_selection = 25 * Hz
stimuli1 = TimedArray(np.r_[np.zeros(40), np.ones(2), np.zeros(100)], dt=25 * ms)
input1 = PoissonInput(P_E[N_non:N_non + N_sub], 's_AMPA_ext', C_selection, rate_
↪selection, 'stimuli1(t)')
```

```python
# at 2s, select population 2
stimuli2 = TimedArray(np.r_[np.zeros(80), np.ones(2), np.zeros(100)], dt=25 * ms)
input2 = PoissonInput(P_E[N_non + N_sub:N_non + 2 * N_sub], 's_AMPA_ext', C_selection,
↪ rate_selection, 'stimuli2(t)')

# at 4s, reset selection
stimuli_reset = TimedArray(np.r_[np.zeros(120), np.ones(2), np.zeros(100)], dt=25 *
↪ms)
input_reset_I = PoissonInput(P_E, 's_AMPA_ext', C_ext, rate_selection, 'stimuli_
↪reset(t)')
input_reset_E = PoissonInput(P_I, 's_AMPA_ext', C_ext, rate_selection, 'stimuli_
↪reset(t)')

# monitors
N_activity_plot = 15
sp_E_sels = [SpikeMonitor(P_E[pi:pi + N_activity_plot]) for pi in range(N_non, N_non
↪+ p * N_sub, N_sub)]
sp_E = SpikeMonitor(P_E[:N_activity_plot])
sp_I = SpikeMonitor(P_I[:N_activity_plot])

r_E_sels = [PopulationRateMonitor(P_E[pi:pi + N_sub]) for pi in range(N_non, N_non +
↪p * N_sub, N_sub)]
r_E = PopulationRateMonitor(P_E[:N_non])
r_I = PopulationRateMonitor(P_I)

# simulate, can be long >120s
net = Network(collect())
net.add(sp_E_sels)
net.add(r_E_sels)
net.run(4 * second, report='stdout')

# plotting
title('Population rates')
xlabel('ms')
ylabel('Hz')

plot(r_E.t / ms, r_E.smooth_rate(width=25 * ms) / Hz, label='nonselective')
plot(r_I.t / ms, r_I.smooth_rate(width=25 * ms) / Hz, label='inhibitory')

for i, r_E_sel in enumerate(r_E_sels[::-1]):
    plot(r_E_sel.t / ms, r_E_sel.smooth_rate(width=25 * ms) / Hz,
        label=f"selective {p - i}")

legend()
figure()

title(f"Population activities ({N_activity_plot} neurons/pop)")
xlabel('ms')
yticks([])

plot(sp_E.t / ms, sp_E.i + (p + 1) * N_activity_plot, '.', markersize=2,
    label="nonselective")
plot(sp_I.t / ms, sp_I.i + p * N_activity_plot, '.', markersize=2, label="inhibitory")

for i, sp_E_sel in enumerate(sp_E_sels[::-1]):
    plot(sp_E_sel.t / ms, sp_E_sel.i + (p - i - 1) * N_activity_plot, '.',
↪markersize=2,
```

```
        label=f"selective {p - i}")

legend()
show()
```


Population rates

Population activities (15 neurons/pop)
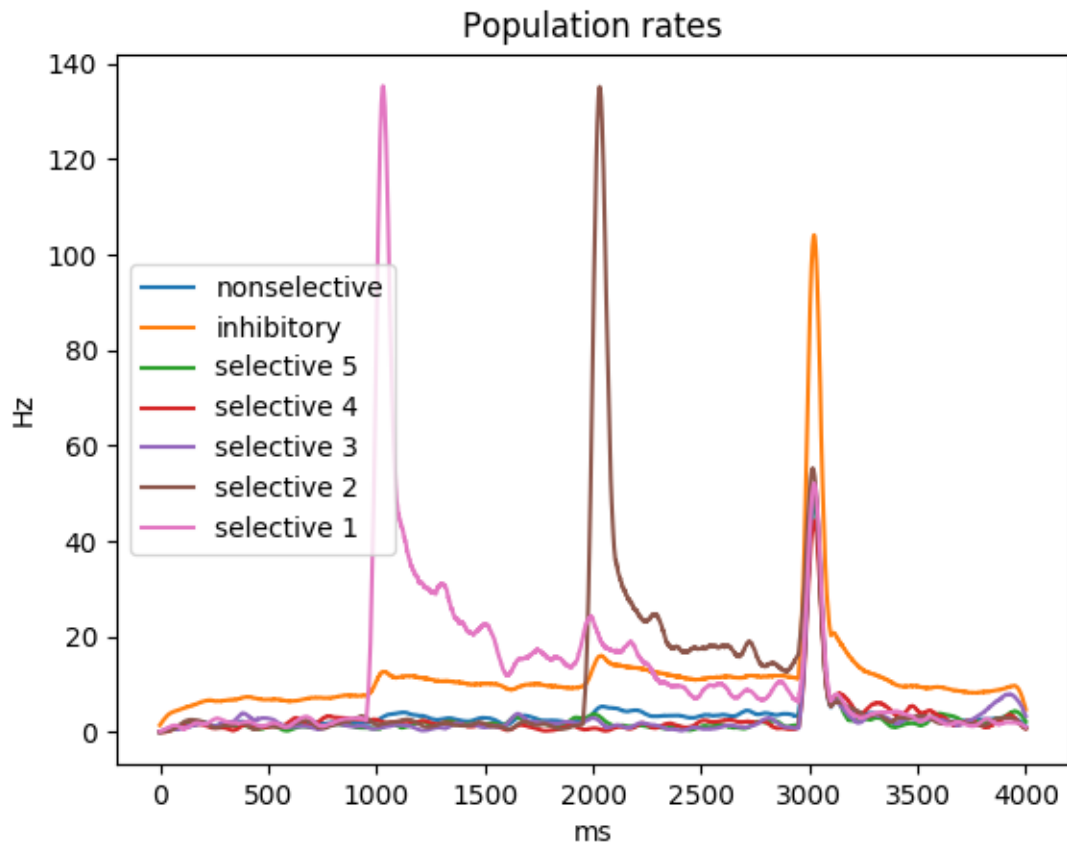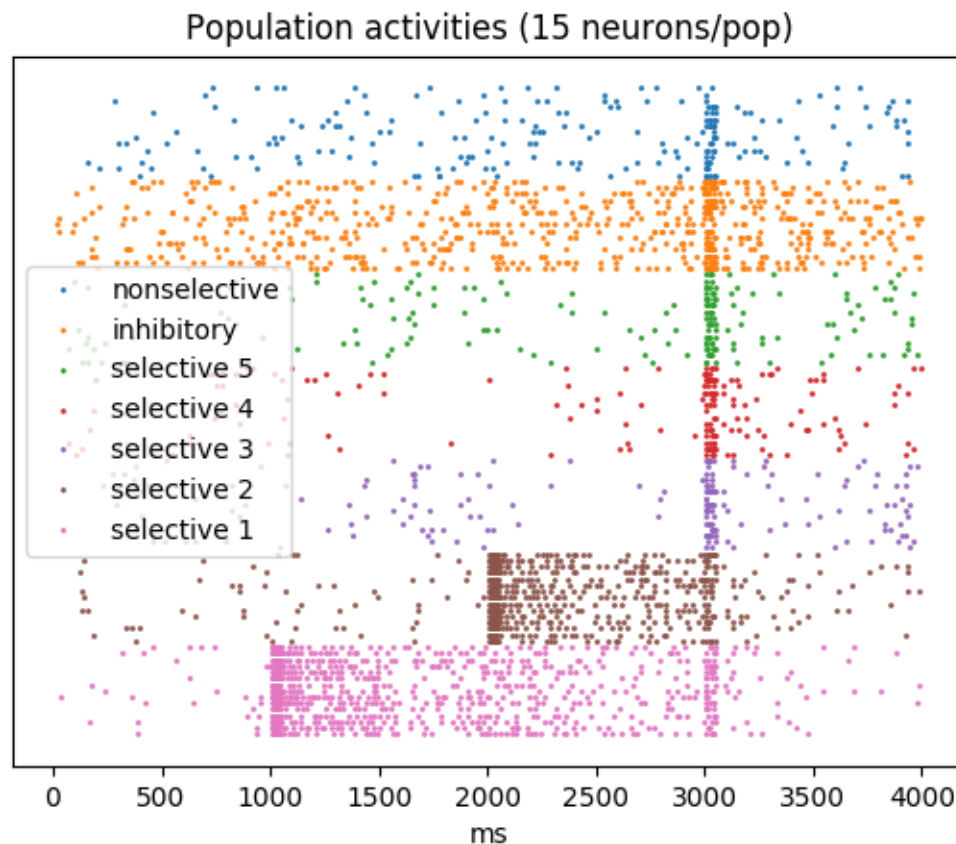
5.11.6 Example: Clopath_et_al_2010_homeostasis

This code contains an adapted version of the voltage-dependent triplet STDP rule from: Clopath et al., Connectivity reflects coding: a model of voltage-based STDP with homeostasis, Nature Neuroscience, 2010 (http://dx.doi.org/10.1038/nn.2479)

The plasticity rule is adapted for a leaky integrate & fire model in Brian2. More specifically, the filters `v_lowpass1` and `v_lowpass2` are incremented by a constant at every post-synaptic spike time, to compensate for the lack of an actual spike in the integrate & fire model.

As an illustration of the rule, we simulate the competition between inputs projecting on a downstream neuron. We would like to note that the parameters have been chosen arbitrarily to qualitatively reproduce the behavior of the original work, but need additional fitting.

We kindly ask to cite the article when using the model presented below.

This code was written by Jacopo Bono, 12/2015

```python
from brian2 import *

#######################################################################
# PLASTICITY MODEL
#######################################################################
```

(continues on next page)

```
#### Plasticity Parameters

V_rest = -70.*mV          # resting potential
V_thresh = -55.*mV        # spiking threshold
Theta_low = V_rest        # depolarization threshold for plasticity
x_reset = 1.              # spike trace reset value
taux = 15.*ms             # spike trace time constant
A_LTD = 1.5e-4            # depression amplitude
A_LTP = 1.5e-2            # potentiation amplitude
tau_lowpass1 = 40*ms      # timeconstant for low-pass filtered voltage
tau_lowpass2 = 30*ms      # timeconstant for low-pass filtered voltage
tau_homeo = 1000*ms       # homeostatic timeconstant
v_target = 12*mV**2       # target depolarisation


#### Plasticity Equations

# equations executed at every timestepC
Syn_model =   ('''
            w_ampa:1                    # synaptic weight (ampa synapse)
            ''')

# equations executed only when a presynaptic spike occurs
Pre_eq = ('''
            g_ampa_post += w_ampa*ampa_max_cond                              ␣
↪                        # increment synaptic conductance
            A_LTD_u = A_LTD*(v_homeo**2/v_target)                            ␣
↪                        # metaplasticity
            w_minus = A_LTD_u*(v_lowpass1_post/mV - Theta_low/mV)*int(v_lowpass1_post/
↪mV - Theta_low/mV > 0)   # synaptic depression
            w_ampa = clip(w_ampa-w_minus, 0, w_max)                         ␣
↪                        # hard bounds
            ''' )

# equations executed only when a postsynaptic spike occurs
Post_eq = ('''
            v_lowpass1 += 10*mV                                              ␣
↪                           # mimics the depolarisation effect due to a spike
            v_lowpass2 += 10*mV                                              ␣
↪                           # mimics the depolarisation effect due to a spike
            v_homeo += 0.1*mV                                               ␣
↪                           # mimics the depolarisation effect due to a spike
            w_plus = A_LTP*x_trace_pre*(v_lowpass2_post/mV - Theta_low/mV)*int(v_
↪lowpass2_post/mV - Theta_low/mV > 0)  # synaptic potentiation
            w_ampa = clip(w_ampa+w_plus, 0, w_max)                          ␣
↪                           # hard bounds
            ''' )


###############################################################################
# I&F Parameters and equations
###############################################################################


#### Neuron parameters

gleak = 30.*nS                    # leak conductance
C = 300.*pF                       # membrane capacitance
tau_AMPA = 2.*ms                  # AMPA synaptic timeconstant
```

```
E_AMPA = 0.*mV                     # reversal potential AMPA

ampa_max_cond = 5.e-8*siemens   # Ampa maximal conductance
w_max = 1.                         # maximal ampa weight


#### Neuron Equations

# We connect 10 presynaptic neurons to 1 downstream neuron

# downstream neuron
eqs_neurons = '''
dv/dt = (gleak*(V_rest-v) + I_ext + I_syn)/C: volt       # voltage
dv_lowpass1/dt = (v-v_lowpass1)/tau_lowpass1 : volt      # low-pass filter of the
↪voltage
dv_lowpass2/dt = (v-v_lowpass2)/tau_lowpass2 : volt      # low-pass filter of the
↪voltage
dv_homeo/dt = (v-V_rest-v_homeo)/tau_homeo : volt        # low-pass filter of the
↪voltage
I_ext : amp                                             # external current
I_syn = g_ampa*(E_AMPA-v): amp                          # synaptic current
dg_ampa/dt = -g_ampa/tau_AMPA : siemens                 # synaptic conductance
dx_trace/dt = -x_trace/taux :1                          # spike trace
'''

# input neurons
eqs_inputs = '''
dv/dt = gleak*(V_rest-v)/C: volt                        # voltage
dx_trace/dt = -x_trace/taux :1                          # spike trace
rates : Hz                                              # input rates
selected_index : integer (shared)                      # active neuron
'''


##############################################################################
# Simulation
##############################################################################

#### Parameters

defaultclock.dt = 500.*us                          # timestep
Nr_neurons = 1                                      # Number of downstream neurons
Nr_inputs = 5                                       # Number of input neurons
input_rate = 35*Hz                                  # Rates
init_weight = 0.5                                   # initial synaptic weight
final_t = 20.*second                               # end of simulation
input_time = 100.*ms                               # duration of an input

#### Create neuron objects

Nrn_downstream = NeuronGroup(Nr_neurons, eqs_neurons, threshold='v>V_thresh',
                        reset='v=V_rest;x_trace+=x_reset/(taux/ms)',
                        method='euler')
Nrns_input = NeuronGroup(Nr_inputs, eqs_inputs, threshold='rand()<rates*dt',
                    reset='v=V_rest;x_trace+=x_reset/(taux/ms)',
                    method='exact')

#### create Synapses
```

```python
Syn = Synapses(Nrns_input, Nrn_downstream,
               model=Syn_model,
               on_pre=Pre_eq,
               on_post=Post_eq
               )

Syn.connect(i=numpy.arange(Nr_inputs), j=0)

#### Monitors and storage
W_evolution = StateMonitor(Syn, 'w_ampa', record=True)

#### Run

# Initial values
Nrn_downstream.v = V_rest
Nrn_downstream.v_lowpass1 = V_rest
Nrn_downstream.v_lowpass2 = V_rest
Nrn_downstream.v_homeo = 0
Nrn_downstream.I_ext = 0.*amp
Nrn_downstream.x_trace = 0.
Nrns_input.v = V_rest
Nrns_input.x_trace = 0.
Syn.w_ampa = init_weight

# Switch on a different input every 100ms
Nrns_input.run_regularly('''
                         selected_index = int(floor(rand()*Nr_inputs))
                         rates = input_rate * int(selected_index == i)  # All rates␣
→are zero except for the selected neuron
                         ''', dt=input_time)
run(final_t, report='text')

###########################################################################
# Plots
###########################################################################
stitle = 'Synaptic Competition'

fig = figure(figsize=(8, 5))
for kk in range(Nr_inputs):
    plt.plot(W_evolution.t, W_evolution.w_ampa[kk], '-', linewidth=2)
xlabel('Time [ms]', fontsize=22)
ylabel('Weight [a.u.]', fontsize=22)
plt.subplots_adjust(bottom=0.2, left=0.15, right=0.95, top=0.85)
title(stitle, fontsize=22)
plt.show()
```

Synaptic Competition
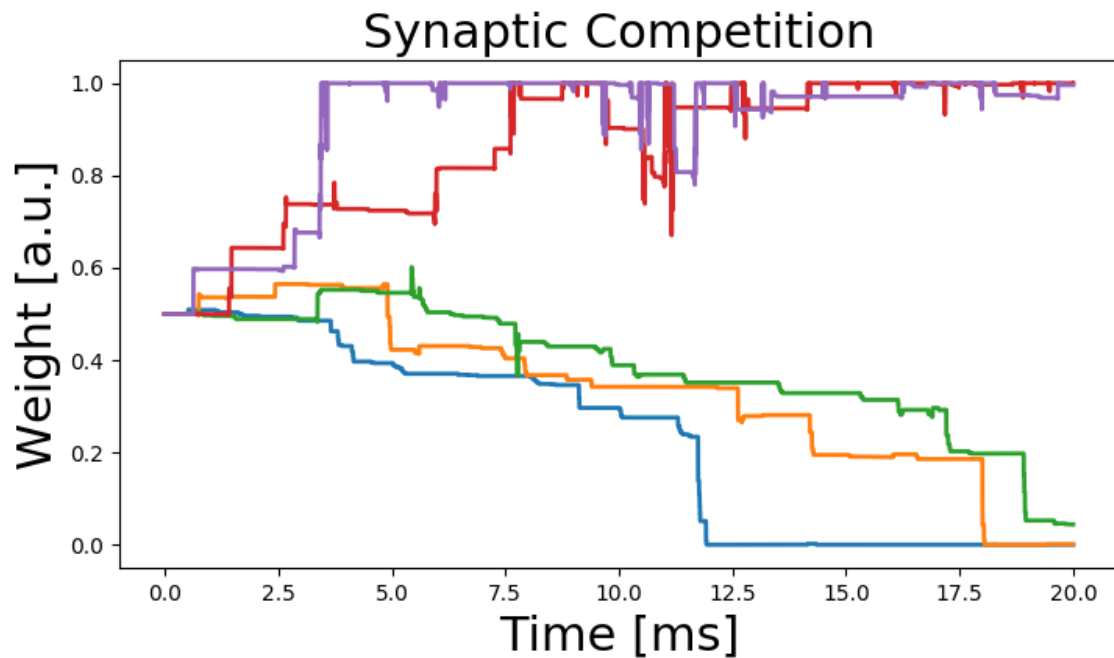
5.11.7 Example: Clopath_et_al_2010_no_homeostasis

This code contains an adapted version of the voltage-dependent triplet STDP rule from: Clopath et al., Connectivity reflects coding: a model of voltage-based STDP with homeostasis, Nature Neuroscience, 2010 (http://dx.doi.org/10.1038/nn.2479)

The plasticity rule is adapted for a leaky integrate & fire model in Brian2. In particular, the filters `v_lowpass1` and `v_lowpass2` are incremented by a constant at every post-synaptic spike time, to compensate for the lack of an actual spike in the integrate & fire model. Moreover, this script does not include the homeostatic metaplasticity.

As an illustration of the Rule, we simulate a plot analogous to figure 2b in the above article, showing the frequency dependence of plasticity as measured in: Sjöström et al., Rate, timing and cooperativity jointly determine cortical synaptic plasticity. Neuron, 2001. We would like to note that the parameters have been chosen arbitrarily to qualitatively reproduce the behavior of the original works, but need additional fitting.

We kindly ask to cite both articles when using the model presented below.

This code was written by Jacopo Bono, 12/2015

```python
from brian2 import *
#########################################################################
# PLASTICITY MODEL
#########################################################################


#### Plasticity Parameters

V_rest = -70.*mV          # resting potential
V_thresh = -50.*mV        # spiking threshold
```

(continues on next page)

```
Theta_low = V_rest          # depolarization threshold for plasticity
x_reset = 1.                # spike trace reset value
taux = 15.*ms               # spike trace time constant
A_LTD = 1.5e-4              # depression amplitude
A_LTP = 1.5e-2              # potentiation amplitude
tau_lowpass1 = 40*ms        # timeconstant for low-pass filtered voltage
tau_lowpass2 = 30*ms        # timeconstant for low-pass filtered voltage



#### Plasticity Equations


# equations executed at every timestep
Syn_model = '''
            w_ampa:1                    # synaptic weight (ampa synapse)
            '''

# equations executed only when a presynaptic spike occurs
Pre_eq = '''
        g_ampa_post += w_ampa*ampa_max_cond                                    ␣
↪                  # increment synaptic conductance
        w_minus = A_LTD*(v_lowpass1_post/mV - Theta_low/mV)*int(v_lowpass1_post/mV -␣
↪Theta_low/mV > 0)  # synaptic depression
        w_ampa = clip(w_ampa-w_minus,0,w_max)                                  ␣
↪                  # hard bounds
        '''


# equations executed only when a postsynaptic spike occurs
Post_eq = '''
        v_lowpass1 += 10*mV                                                    ␣
↪                          # mimics the depolarisation by a spike
        v_lowpass2 += 10*mV                                                    ␣
↪                          # mimics the depolarisation by a spike
        w_plus = A_LTP*x_trace_pre*(v_lowpass2_post/mV - Theta_low/mV)*int(v_
↪lowpass2_post/mV - Theta_low/mV > 0)  # synaptic potentiation
        w_ampa = clip(w_ampa+w_plus,0,w_max)                                   ␣
↪                          # hard bounds
        '''


################################################################################
# I&F Parameters and equations
################################################################################

#### Neuron parameters

gleak = 30.*nS                   # leak conductance
C = 300.*pF                      # membrane capacitance
tau_AMPA = 2.*ms                 # AMPA synaptic timeconstant
E_AMPA = 0.*mV                   # reversal potential AMPA

ampa_max_cond = 5.e-10*siemens   # Ampa maximal conductance
w_max = 1.                       # maximal ampa weight


#### Neuron Equations
```

```
eqs_neurons = '''
dv/dt = (gleak*(V_rest-v) + I_ext + I_syn)/C: volt      # voltage
dv_lowpass1/dt = (v-v_lowpass1)/tau_lowpass1 : volt     # low-pass filter of the
↪voltage
dv_lowpass2/dt = (v-v_lowpass2)/tau_lowpass2 : volt     # low-pass filter of the
↪voltage
I_ext : amp                                             # external current
I_syn = g_ampa*(E_AMPA-v): amp                          # synaptic current
dg_ampa/dt = -g_ampa/tau_AMPA : siemens                 # synaptic conductance
dx_trace/dt = -x_trace/taux :1                          # spike trace
'''




###############################################################################
# Simulation
###############################################################################


#### Parameters

defaultclock.dt = 100.*us                          # timestep
Nr_neurons = 2                                      # Number of neurons
rate_array = [1., 5., 10., 15., 20., 30., 50.]*Hz  # Rates
init_weight = 0.5                                   # initial synaptic weight
reps = 15                                           # Number of pairings


#### Create neuron objects

Nrns = NeuronGroup(Nr_neurons, eqs_neurons, threshold='v>V_thresh',
                   reset='v=V_rest;x_trace+=x_reset/(taux/ms)', method='euler')#

#### create Synapses

Syn = Synapses(Nrns, Nrns,
               model=Syn_model,
               on_pre=Pre_eq,
               on_post=Post_eq
               )

Syn.connect('i!=j')

#### Monitors and storage
weight_result = np.zeros((2, len(rate_array)))              # to save the final
↪weights

#### Run

# loop over rates
for jj, rate in enumerate(rate_array):

    # Calculate interval between pairs
    pair_interval = 1./rate - 10*ms
    print('Starting simulations for %s' % rate)

    # Initial values
    Nrns.v = V_rest
    Nrns.v_lowpass1 = V_rest
```

```
    Nrns.v_lowpass2 = V_rest
    Nrns.I_ext = 0*amp
    Nrns.x_trace = 0.
    Syn.w_ampa = init_weight

    # loop over pairings
    for ii in range(reps):
        # 1st SPIKE
        Nrns.v[0] = V_thresh + 1*mV
        # 2nd SPIKE
        run(10*ms)
        Nrns.v[1] = V_thresh + 1*mV
        # run
        run(pair_interval)
        print('Pair %d out of %d' % (ii+1, reps))

    #store weight changes
    weight_result[0, jj] = 100.*Syn.w_ampa[0]/init_weight
    weight_result[1, jj] = 100.*Syn.w_ampa[1]/init_weight


###############################################################################
# Plots
###############################################################################

stitle = 'Pairings'
scolor = 'k'

figure(figsize=(8, 5))
plot(rate_array, weight_result[0,:], '-', linewidth=2, color=scolor)
plot(rate_array, weight_result[1,:], ':', linewidth=2, color=scolor)
xlabel('Pairing frequency [Hz]', fontsize=22)
ylabel('Normalised Weight [%]', fontsize=22)
legend(['Pre-Post', 'Post-Pre'], loc='best')
subplots_adjust(bottom=0.2, left=0.15, right=0.95, top=0.85)
title(stitle)
show()
```
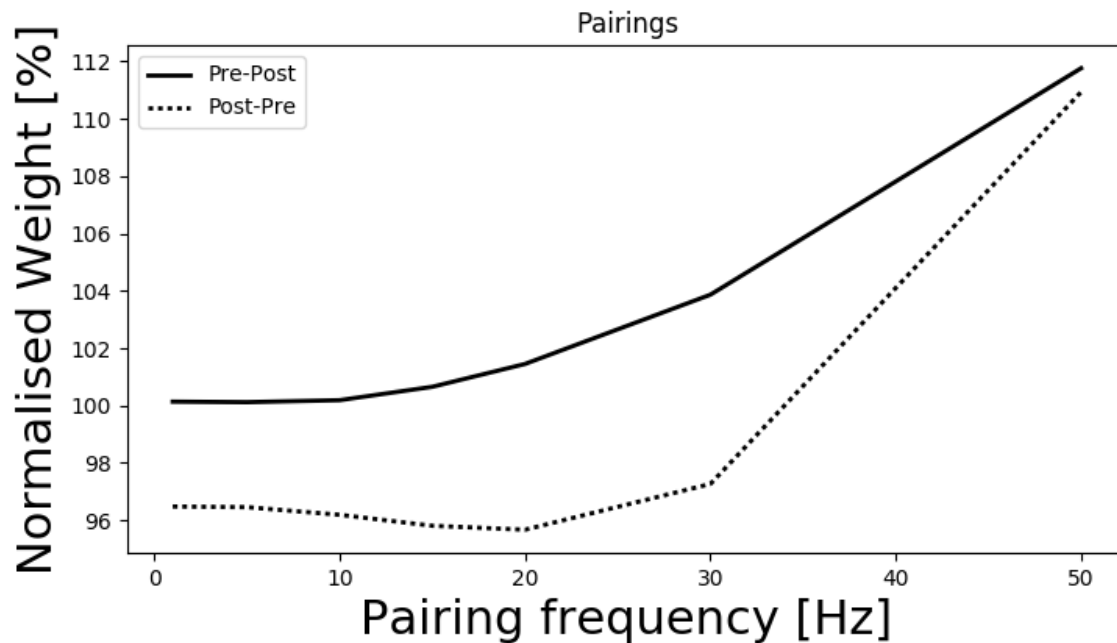
5.11.8 Example: Destexhe_et_al_1998

Reproduces Figure 12 (simplified three-compartment model) from the following paper:

Dendritic Low-Threshold Calcium Currents in Thalamic Relay Cells Alain Destexhe, Mike Neubig, Daniel Ulrich, John Huguenard Journal of Neuroscience 15 May 1998, 18 (10) 3574-3588

The original NEURON code is available on ModelDB: https://senselab.med.yale.edu/modeldb/ShowModel.cshtml?model=279

Reference for the original morphology:

Rat VB neuron (thalamocortical cell), given by J. Huguenard, stained with biocytin and traced by A. Destexhe, December 1992. The neuron is described in: J.R. Huguenard & D.A. Prince, A novel T-type current underlies prolonged calcium-dependent burst firing in GABAergic neurons of rat thalamic reticular nucleus. J. Neurosci. 12: 3804-3817, 1992.

Available at NeuroMorpho.org:

http://neuromorpho.org/neuron_info.jsp?neuron_name=tc200 NeuroMorpho.Org ID :NMO_00881

Notes

- Completely removed the "Fast mechanism for submembranal Ca++ concentration (cai)" – it did not affect the results presented here

- Time constants for the I_T current are slightly different from the equations given in the paper – the paper calculation seems to be based on 36 degree Celsius but the temperature that is used is 34 degrees.

- To reproduce Figure 12C, the "presence of dendritic shunt conductances" meant setting g_L to 0.15 mS/cm^2 for the whole neuron.

- Other small discrepancies with the paper – values from the NEURON code were used whenever different from the values stated in the paper

```python
from brian2 import *
from brian2.units.constants import (zero_celsius, faraday_constant as F,
                                    gas_constant as R)


defaultclock.dt = 0.01*ms


VT = -52*mV
El = -76.5*mV  # from code, text says: -69.85*mV


E_Na = 50*mV
E_K = -100*mV
C_d = 7.954   # dendritic correction factor


T = 34*kelvin + zero_celsius # 34 degC (current-clamp experiments)
tadj_HH = 3.0**((34-36)/10.0)   # temperature adjustment for Na & K (original
→recordings at 36 degC)
tadj_m_T = 2.5**((34-24)/10.0)
tadj_h_T = 2.5**((34-24)/10.0)


shift_I_T = -1*mV


gamma = F/(R*T)   # R=gas constant, F=Faraday constant
Z_Ca = 2   # Valence of Calcium ions
Ca_i = 240*nM   # intracellular Calcium concentration
Ca_o = 2*mM   # extracellular Calcium concentration


eqs = Equations('''
Im = gl*(El-v) - I_Na - I_K - I_T: amp/meter**2
I_inj : amp (point current)
gl : siemens/meter**2

# HH-type currents for spike initiation
g_Na : siemens/meter**2
g_K : siemens/meter**2
I_Na = g_Na * m**3 * h * (v-E_Na) : amp/meter**2
I_K = g_K * n**4 * (v-E_K) : amp/meter**2
v2 = v - VT : volt  # shifted membrane potential (Traub convention)
dm/dt = (0.32*(mV**-1)*(13.*mV-v2)/
        (exp((13.*mV-v2)/(4.*mV))-1.)*(1-m)-0.28*(mV**-1)*(v2-40.*mV)/
        (exp((v2-40.*mV)/(5.*mV))-1.)*m) / ms * tadj_HH: 1
dn/dt = (0.032*(mV**-1)*(15.*mV-v2)/
        (exp((15.*mV-v2)/(5.*mV))-1.)*(1.-n)-.5*exp((10.*mV-v2)/(40.*mV))*n) / ms *
→tadj_HH: 1
dh/dt = (0.128*exp((17.*mV-v2)/(18.*mV))*(1.-h)-4./(1+exp((40.*mV-v2)/(5.*mV)))*h) /
→ms * tadj_HH: 1
```

(continues on next page)

```python
# Low-threshold Calcium current (I_T)  -- nonlinear function of voltage
I_T = P_Ca * m_T**2*h_T * G_Ca : amp/meter**2
P_Ca : meter/second  # maximum Permeability to Calcium
G_Ca = Z_Ca**2*F*v*gamma*(Ca_i - Ca_o*exp(-Z_Ca*gamma*v))/(1 - exp(-Z_Ca*gamma*v)) :␣
↪coulomb/meter**3
dm_T/dt = -(m_T - m_T_inf)/tau_m_T : 1
dh_T/dt = -(h_T - h_T_inf)/tau_h_T : 1
m_T_inf = 1/(1 + exp(-(v/mV + 56)/6.2)) : 1
h_T_inf = 1/(1 + exp((v/mV + 80)/4)) : 1
tau_m_T = (0.612 + 1.0/(exp(-(v/mV + 131)/16.7) + exp((v/mV + 15.8)/18.2))) * ms /␣
↪tadj_m_T: second
tau_h_T = (int(v<-81*mV) * exp((v/mV + 466)/66.6) +
           int(v>=-81*mV) * (28 + exp(-(v/mV + 21)/10.5))) * ms / tadj_h_T: second
''')


# Simplified three-compartment morphology
morpho = Cylinder(x=[0, 38.42]*um, diameter=26*um)
morpho.dend = Cylinder(x=[0, 12.49]*um, diameter=10.28*um)
morpho.dend.distal = Cylinder(x=[0, 84.67]*um, diameter=8.5*um)
neuron = SpatialNeuron(morpho, eqs, Cm=0.88*uF/cm**2, Ri=173*ohm*cm,
                       method='exponential_euler')


neuron.v = -74*mV
# Only the soma has Na/K channels
neuron.main.g_Na = 100*msiemens/cm**2
neuron.main.g_K = 100*msiemens/cm**2
# Apply the correction factor to the dendrites

neuron.dend.Cm *= C_d
neuron.m_T = 'm_T_inf'
neuron.h_T = 'h_T_inf'

mon = StateMonitor(neuron, ['v'], record=True)

store('initial state')


def do_experiment(currents, somatic_density, dendritic_density,
                  dendritic_conductance=0.0379*msiemens/cm**2,
                  HH_currents=True):
    restore('initial state')
    voltages = []
    neuron.P_Ca = somatic_density
    neuron.dend.distal.P_Ca = dendritic_density * C_d
    # dendritic conductance (shunting conductance used for Fig 12C)
    neuron.gl = dendritic_conductance
    neuron.dend.gl = dendritic_conductance * C_d
    if not HH_currents:
        # Shut off spiking (for Figures 12B and 12C)
        neuron.g_Na = 0*msiemens/cm**2
        neuron.g_K = 0*msiemens/cm**2
    run(180*ms)
    store('before current')
    for current in currents:
        restore('before current')
        neuron.main.I_inj = current
```

```
        print('.', end='')
        run(320*ms)
        voltages.append(mon[morpho].v[:])  # somatic voltage
    return voltages


## Run the various variants of the model to reproduce Figure 12
mpl.rcParams['lines.markersize'] = 3.0
fig, axes = plt.subplots(2, 2)
print('Running experiments for Figure A1 ', end='')
voltages = do_experiment([50, 75]*pA, somatic_density=1.7e-5*cm/second,
                         dendritic_density=1.7e-5*cm/second)
print(' done.')
cut_off = 100*ms  # Do not display first part of simulation
axes[0, 0].plot((mon.t - cut_off) / ms, voltages[0] / mV, color='gray')
axes[0, 0].plot((mon.t - cut_off) / ms, voltages[1] / mV, color='black')
axes[0, 0].set(xlim=(0, 400), ylim=(-80, 40), xticks=[],
               title='A1: Uniform T-current density', ylabel='Voltage (mV)')
axes[0, 0].spines['right'].set_visible(False)
axes[0, 0].spines['top'].set_visible(False)
axes[0, 0].spines['bottom'].set_visible(False)


print('Running experiments for Figure A2 ', end='')
voltages = do_experiment([50, 75]*pA, somatic_density=1.7e-5*cm/second,
                         dendritic_density=9.5e-5*cm/second)
print(' done.')
cut_off = 100*ms  # Do not display first part of simulation
axes[1, 0].plot((mon.t - cut_off) / ms, voltages[0] / mV, color='gray')
axes[1, 0].plot((mon.t - cut_off) / ms, voltages[1] / mV, color='black')
axes[1, 0].set(xlim=(0, 400), ylim=(-80, 40),
               title='A2: High T-current density in dendrites',
               xlabel='Time (ms)', ylabel='Voltage (mV)')
axes[1, 0].spines['right'].set_visible(False)
axes[1, 0].spines['top'].set_visible(False)


print('Running experiments for Figure B ', end='')
currents = np.linspace(0, 200, 41)*pA
voltages_somatic = do_experiment(currents, somatic_density=56.36e-5*cm/second,
                                 dendritic_density=0*cm/second,
                                 HH_currents=False)
voltages_somatic_dendritic = do_experiment(currents, somatic_density=1.7e-5*cm/second,
                                           dendritic_density=9.5e-5*cm/second,
                                           HH_currents=False)
print(' done.')
maxima_somatic = Quantity(voltages_somatic).max(axis=1)
maxima_somatic_dendritic = Quantity(voltages_somatic_dendritic).max(axis=1)
axes[0, 1].yaxis.tick_right()
axes[0, 1].plot(currents/pA, maxima_somatic/mV,
                'o-', color='black', label='Somatic only')
axes[0, 1].plot(currents/pA, maxima_somatic_dendritic/mV,
                's-', color='black', label='Somatic & dendritic')
axes[0, 1].set(xlabel='Injected current (pA)', ylabel='Peak LTS (mV)',
               ylim=(-80, 0))
axes[0, 1].legend(loc='best', frameon=False)

print('Running experiments for Figure C ', end='')
currents = np.linspace(200, 400, 41)*pA
```

```
voltages_somatic = do_experiment(currents, somatic_density=56.36e-5*cm/second,
                                 dendritic_density=0*cm/second,
                                 dendritic_conductance=0.15*msiemens/cm**2,
                                 HH_currents=False)
voltages_somatic_dendritic = do_experiment(currents, somatic_density=1.7e-5*cm/second,
                                           dendritic_density=9.5e-5*cm/second,
                                           dendritic_conductance=0.15*msiemens/cm**2,
                                           HH_currents=False)
print(' done.')
maxima_somatic = Quantity(voltages_somatic).max(axis=1)
maxima_somatic_dendritic = Quantity(voltages_somatic_dendritic).max(axis=1)
axes[1, 1].yaxis.tick_right()
axes[1, 1].plot(currents/pA, maxima_somatic/mV,
                'o-', color='black', label='Somatic only')
axes[1, 1].plot(currents/pA, maxima_somatic_dendritic/mV,
                's-', color='black', label='Somatic & dendritic')
axes[1, 1].set(xlabel='Injected current (pA)', ylabel='Peak LTS (mV)',
               ylim=(-80, 0))
axes[1, 1].legend(loc='best', frameon=False)

plt.tight_layout()
plt.show()
```
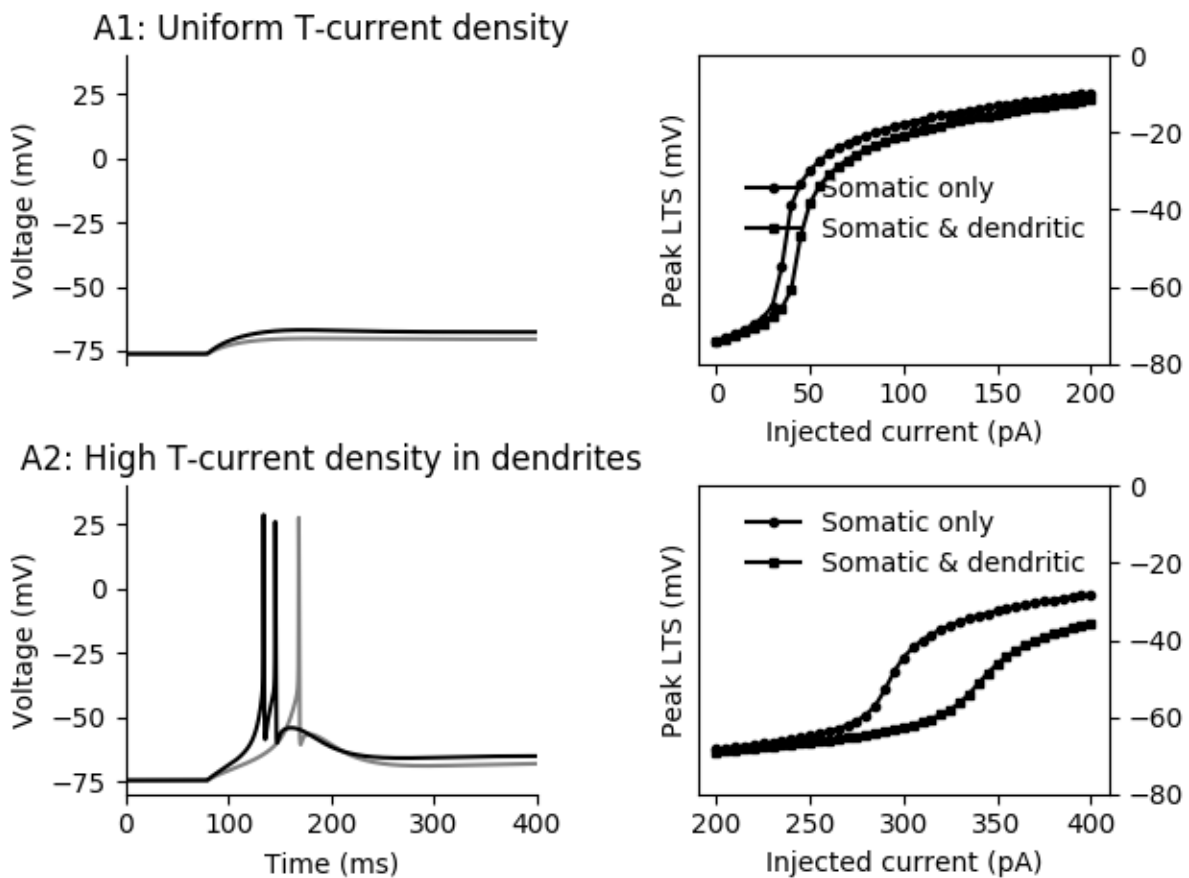
5.11.9 Example: Diesmann_et_al_1999

Synfire chains

M. Diesmann et al. (1999). Stable propagation of synchronous spiking in cortical neural networks. Nature 402, 529-533.

```python
from brian2 import *

duration = 100*ms

# Neuron model parameters
Vr = -70*mV
Vt = -55*mV
taum = 10*ms
taupsp = 0.325*ms
weight = 4.86*mV
# Neuron model
eqs = Equations('''
dV/dt = (-(V-Vr)+x)*(1./taum) : volt
dx/dt = (-x+y)*(1./taupsp) : volt
dy/dt = -y*(1./taupsp)+25.27*mV/ms+
        (39.24*mV/ms**0.5)*xi : volt
''')

# Neuron groups
n_groups = 10
group_size = 100
P = NeuronGroup(N=n_groups*group_size, model=eqs,
                threshold='V>Vt', reset='V=Vr', refractory=1*ms,
                method='euler')

Pinput = SpikeGeneratorGroup(85, np.arange(85),
                             np.random.randn(85)*1*ms + 50*ms)
# The network structure
S = Synapses(P, P, on_pre='y+=weight')
S.connect(j='k for k in range((int(i/group_size)+1)*group_size, (int(i/group_
→size)+2)*group_size) '
          'if i<N_pre-group_size')
Sinput = Synapses(Pinput, P[:group_size], on_pre='y+=weight')
Sinput.connect()

# Record the spikes
Mgp = SpikeMonitor(P)
Minput = SpikeMonitor(Pinput)
# Setup the network, and run it
P.V = 'Vr + rand() * (Vt - Vr)'
run(duration)

plot(Mgp.t/ms, 1.0*Mgp.i/group_size, '.')
plot([0, duration/ms], np.arange(n_groups).repeat(2).reshape(-1, 2).T, 'k-')
ylabel('group number')
yticks(np.arange(n_groups))
xlabel('time (ms)')
show()
```
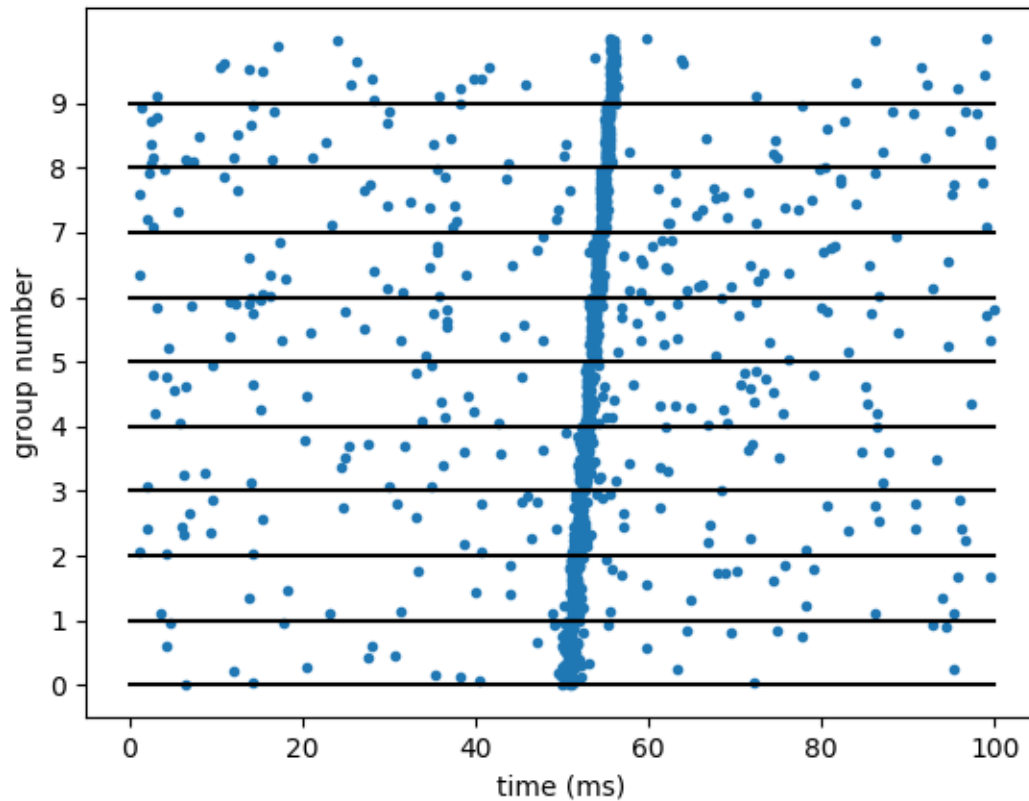
5.11.10 Example: Hindmarsh_Rose_1984

Burst generation in the Hinsmarsh-Rose model. Reproduces Figure 6 of:

Hindmarsh, J. L., and R. M. Rose. "A Model of Neuronal Bursting Using Three Coupled First Order Differential Equations." Proceedings of the Royal Society of London. Series B, Biological Sciences 221, no. 1222 (1984): 87–102.

```python
from brian2 import *

# In the original model, time is measured in arbitrary time units
time_unit = 1*ms
defaultclock.dt = time_unit/10

x_1 = -1.6  # leftmost equilibrium point of the model without adaptation
a = 1; b = 3; c = 1; d = 5
r = 0.001; s = 4
eqs = '''
dx/dt = (y - a*x**3 + b*x**2 + I - z)/time_unit : 1
dy/dt = (c - d*x**2 - y)/time_unit : 1
dz/dt = r*(s*(x - x_1) - z)/time_unit : 1
I : 1 (constant)
'''
```

```python
# We run the model with three different currents
neuron = NeuronGroup(3, eqs, method='rk4')

# Set all variables to their equilibrium point
neuron.x = x_1
neuron.y = 'c - d*x**2'
neuron.z = 'r*(s*(x - x_1))'

# Set the constant current input
neuron.I = [0.4, 2, 4]

# Record the "membrane potential"
mon = StateMonitor(neuron, 'x', record=True)

run(2100*time_unit)

ax_top = plt.subplot2grid((2, 3), (0, 0), colspan=3)
ax_bottom_l = plt.subplot2grid((2, 3), (1, 0), colspan=2)
ax_bottom_r = plt.subplot2grid((2, 3), (1, 2))
for ax in [ax_top, ax_bottom_l, ax_bottom_r]:
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.set(ylim=(-2, 2), yticks=[-2, 0, 2])

ax_top.plot(mon.t/time_unit, mon.x[0])

ax_bottom_l.plot(mon.t/time_unit, mon.x[1])
ax_bottom_l.set_xlim(700, 2100)

ax_bottom_r.plot(mon.t/time_unit, mon.x[2])
ax_bottom_r.set_xlim(1400, 2100)
ax_bottom_r.set_yticks([])

plt.show()
```
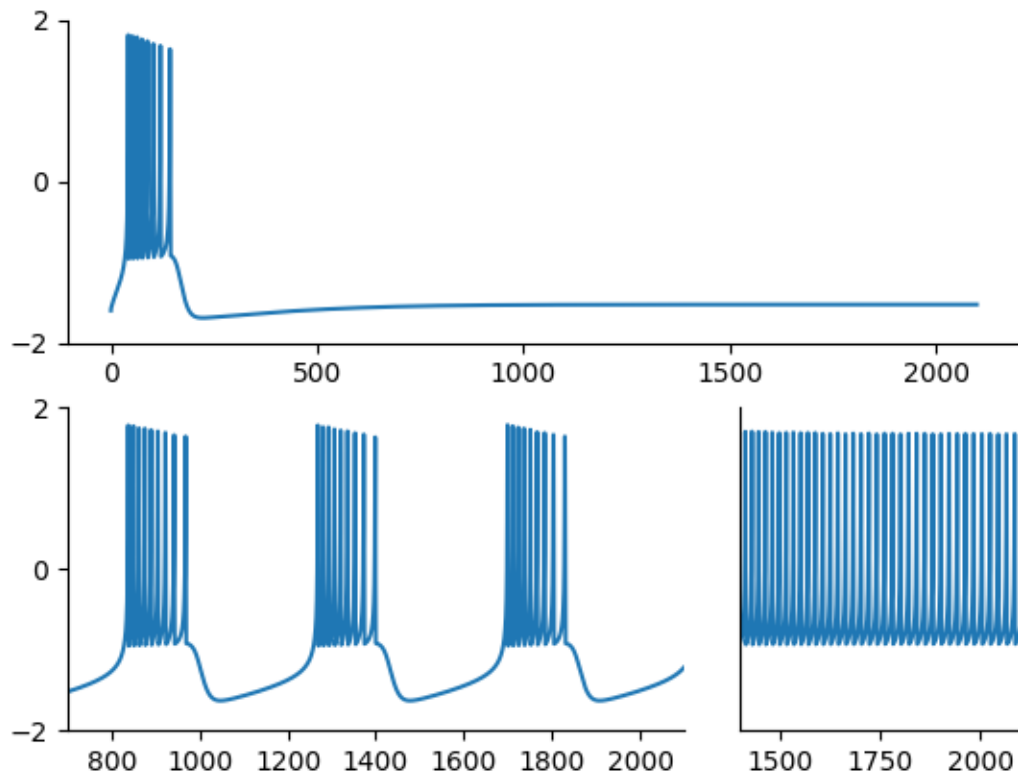
5.11.11 Example: Izhikevich_2007

STDP modulated with reward

Adapted from Fig. 1c of:

Eugene M. Izhikevich Solving the distal reward problem through linkage of STDP and dopamine signaling. Cerebral cortex 17, no. 10 (2007): 2443-2452.

Note: The variable "mode" can switch the behavior of the synapse from "Classical STDP" to "Dopamine modulated STDP".

Author: Guillaume Dumas (Institut Pasteur) Date: 2018-08-24

```python
from brian2 import *

# Parameters
simulation_duration = 6 * second

## Neurons
taum = 10*ms
Ee = 0*mV
vt = -54*mV
vr = -60*mV
```

```python
El = -74*mV
taue = 5*ms

## STDP
taupre = 20*ms
taupost = taupre
gmax = .01
dApre = .01
dApost = -dApre * taupre / taupost * 1.05
dApost *= gmax
dApre *= gmax

## Dopamine signaling
tauc = 1000*ms
taud = 200*ms
taus = 1*ms
epsilon_dopa = 5e-3

# Setting the stage

## Stimuli section
input_indices = array([0, 1, 0, 1, 1, 0,
                       0, 1, 0, 1, 1, 0])
input_times = array([ 500,  550, 1000, 1010, 1500, 1510,
                     3500, 3550, 4000, 4010, 4500, 4510])*ms
spike_input = SpikeGeneratorGroup(2, input_indices, input_times)

neurons = NeuronGroup(2, '''dv/dt = (ge * (Ee-vr) + El - v) / taum : volt
                           dge/dt = -ge / taue : 1''',
                      threshold='v>vt', reset='v = vr',
                      method='exact')
neurons.v = vr
neurons_monitor = SpikeMonitor(neurons)

synapse = Synapses(spike_input, neurons,
                   model='''s: volt''',
                   on_pre='v += s')
synapse.connect(i=[0, 1], j=[0, 1])
synapse.s = 100. * mV

## STDP section
synapse_stdp = Synapses(neurons, neurons,
                   model='''mode: 1
                          dc/dt = -c / tauc : 1 (clock-driven)
                          dd/dt = -d / taud : 1 (clock-driven)
                          ds/dt = mode * c * d / taus : 1 (clock-driven)
                          dApre/dt = -Apre / taupre : 1 (event-driven)
                          dApost/dt = -Apost / taupost : 1 (event-driven)''',
                   on_pre='''ge += s
                           Apre += dApre
                           c = clip(c + mode * Apost, -gmax, gmax)
                           s = clip(s + (1-mode) * Apost, -gmax, gmax)
                           ''',
                   on_post='''Apost += dApost
                           c = clip(c + mode * Apre, -gmax, gmax)
                           s = clip(s + (1-mode) * Apre, -gmax, gmax)
                           ''',
```

```
                    method='euler'
                    )
synapse_stdp.connect(i=0, j=1)
synapse_stdp.mode = 0
synapse_stdp.s = 1e-10
synapse_stdp.c = 1e-10
synapse_stdp.d = 0
synapse_stdp_monitor = StateMonitor(synapse_stdp, ['s', 'c', 'd'], record=[0])

## Dopamine signaling section
dopamine_indices = array([0, 0, 0])
dopamine_times = array([3520, 4020, 4520])*ms
dopamine = SpikeGeneratorGroup(1, dopamine_indices, dopamine_times)
dopamine_monitor = SpikeMonitor(dopamine)
reward = Synapses(dopamine, synapse_stdp, model='''''',
                            on_pre='''d_post += epsilon_dopa''',
                            method='exact')
reward.connect()

# Simulation
## Classical STDP
synapse_stdp.mode = 0
run(simulation_duration/2)
## Dopamine modulated STDP
synapse_stdp.mode = 1
run(simulation_duration/2)

# Visualisation
dopamine_indices, dopamine_times = dopamine_monitor.it
neurons_indices, neurons_times = neurons_monitor.it
figure(figsize=(12, 6))
subplot(411)
plot([0.05, 2.95], [2.7, 2.7], linewidth=5, color='k')
text(1.5, 3, 'Classical STDP', horizontalalignment='center', fontsize=20)
plot([3.05, 5.95], [2.7, 2.7], linewidth=5, color='k')
text(4.5, 3, 'Dopamine modulated STDP', horizontalalignment='center', fontsize=20)
plot(neurons_times, neurons_indices, 'ob')
plot(dopamine_times, dopamine_indices + 2, 'or')
xlim([0, simulation_duration/second])
ylim([-0.5, 4])
yticks([0, 1, 2], ['Pre-neuron', 'Post-neuron', 'Reward'])
xticks([])
subplot(412)
plot(synapse_stdp_monitor.t/second, synapse_stdp_monitor.d.T/gmax, 'r-')
xlim([0, simulation_duration/second])
ylabel('Extracellular\ndopamine d(t)')
xticks([])
subplot(413)
plot(synapse_stdp_monitor.t/second, synapse_stdp_monitor.c.T/gmax, 'b-')
xlim([0, simulation_duration/second])
ylabel('Eligibility\ntrace c(t)')
xticks([])
subplot(414)
plot(synapse_stdp_monitor.t/second, synapse_stdp_monitor.s.T/gmax, 'g-')
xlim([0, simulation_duration/second])
ylabel('Synaptic\nstrength s(t)')
xlabel('Time (s)')
```

```
tight_layout()
show()
```
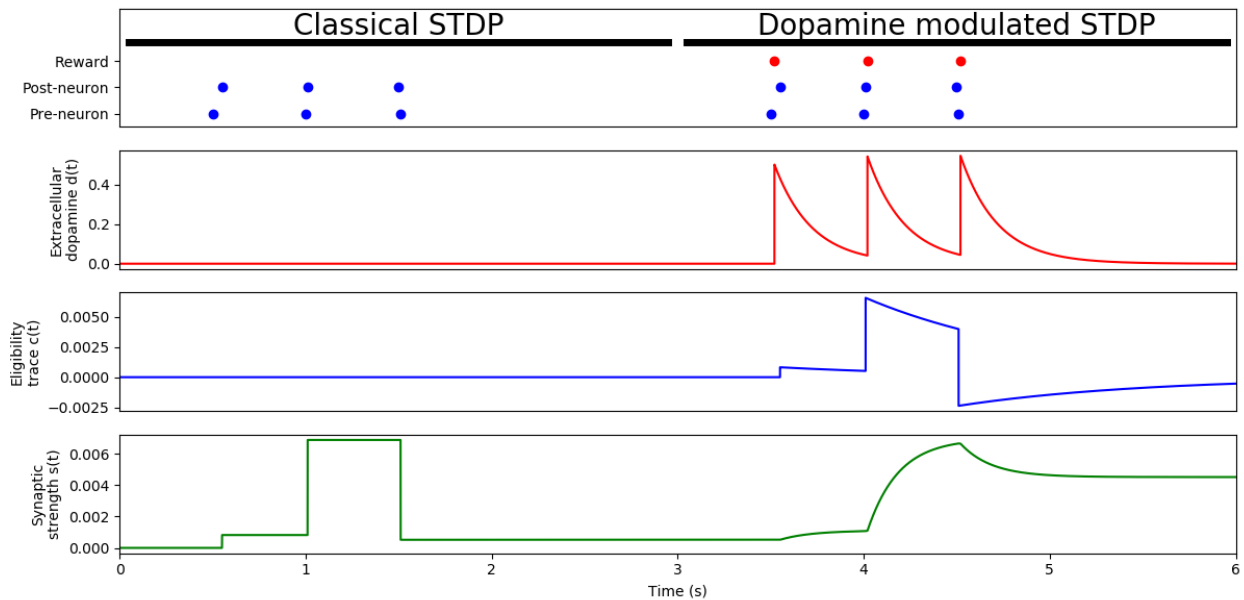

5.11.12 Example: Jansen_Rit_1995_single_column

[Jansen and Rit 1995 model](https://link.springer.com/content/pdf/10.1007/BF00199471.pdf) (Figure 3) in Brian2.

Equations are the system of differential equations number (6) in the original paper. The rate parameters $a=100$ s^{-1} and $b=200$ s^{-1} were changed to excitatory $tau_e = 1000ms/a =10ms$ and inhibitory $tau_i = 1000ms/b =20ms$ time constants as in [Thomas Knosche review](https://link.springer.com/referenceworkentry/10.1007%2F978-1-4614-6675-8_65), [Touboul et al. 2011](https://direct.mit.edu/neco/article-abstract/23/12/3232/7717/Neural-Mass-Activity-Bifurcations-and-Epilepsy?redirectedFrom=fulltext), or [David & Friston 2003](https://www.sciencedirect.com/science/article/pii/S1053811903004579). Units were removerd from parameters e_0, v_0, r_0, A, B, and p to stop Brian's confusion.

Ruben Tikidji-Hamburyan 2021 (rth@r-a-r.org)

```
from numpy import *
from numpy import random as rnd
from matplotlib.pyplot import *
from brian2 import *


defaultclock.dt = .1*ms    #default time step


te,ti    = 10.*ms, 20.*ms #taus for excitatory and inhibitory populations
e0       = 5.             #max firing rate
v0       = 6.             #(max FR)/2 input
r0       = 0.56           #gain rate
A,B,C    = 3.25, 22., 135 #standard parameters as in the set (7) of the original paper
P,deltaP = 120, 320.-120  #random input uniformly distributed between 120 and
                          #320 pulses per second
```

```
# Random noise
nstim = TimedArray(rnd.rand(70000),2*ms)

# Equations as in the system (6) of the original paper
equs = """
dy0/dt = y3                              /second : 1
dy3/dt = (A       * Sp -2*y3 -y0/te*second)/te : 1
dy1/dt = y4                              /second : 1
dy4/dt = (A*(p+ C2 * Se)-2*y4 -y1/te*second)/te : 1
dy2/dt = y5                              /second : 1
dy5/dt = (B   * C4 * Si -2*y5 -y2/ti*second)/ti : 1
p  = P0+nstim(t)*dlP                   : 1
Sp = e0/(1+exp(r0*(v0 - (y1-y2)    ))) : 1
Se = e0/(1+exp(r0*(v0 -  C1*y0     ))) : 1
Si = e0/(1+exp(r0*(v0 -  C3*y0     ))) : 1
C1            : 1
C2 = 0.8 *C1  : 1
C3 = 0.25*C1  : 1
C4 = 0.25*C1  : 1
P0            : 1
dlP           : 1
"""

n = NeuronGroup(6,equs,method='euler') #creates 6 JR models for different␣
↪connectivity parameters

#set parameters as for different traces on figure 3 of the original paper
n.C1[0]   = 68
n.C1[1]   = 128
n.C1[2]   = C
n.C1[3]   = 270
n.C1[4]   = 675
n.C1[5]   = 1350
#set stimulus offset and noise magnitude
n.P0      = P
n.dlP     = deltaP

#just record everything
sm = StateMonitor(n,['y4','y1','y3','y0','y5','y2'],record=True)

#Runs for 5 second
run(5*second,report='text')


#This code goes over all models with different parameters and plot activity of each␣
↪population.

figure(1,figsize=(22,16))
idx1 = where(sm.t/second>2.)[0]


o = 0
for p in [0,1,2,3,4,5]:
    if o == 0: ax = subplot(6,3,1)
    else      :subplot(6,3,1+o,sharex=ax)
    if o == 0: title("E")
    plot(sm.t[idx1]/second, sm[p].y1[idx1],'g-')
```

```
    ylabel(f"C={n[p].C1[0]}")
    if o == 15: xlabel("Time (seconds)")
    subplot(6,3,2+o,sharex=ax)
    if o == 0: title("P")
    plot(sm.t[idx1]/second, sm[p].y0[idx1],'b-')
    if o == 15: xlabel("Time (seconds)")
    subplot(6,3,3+o,sharex=ax)
    if o == 0: title("I")
    plot(sm.t[idx1]/second, sm[p].y2[idx1],'r-')
    if o == 15: xlabel("Time (seconds)")
    o += 3

show()
```
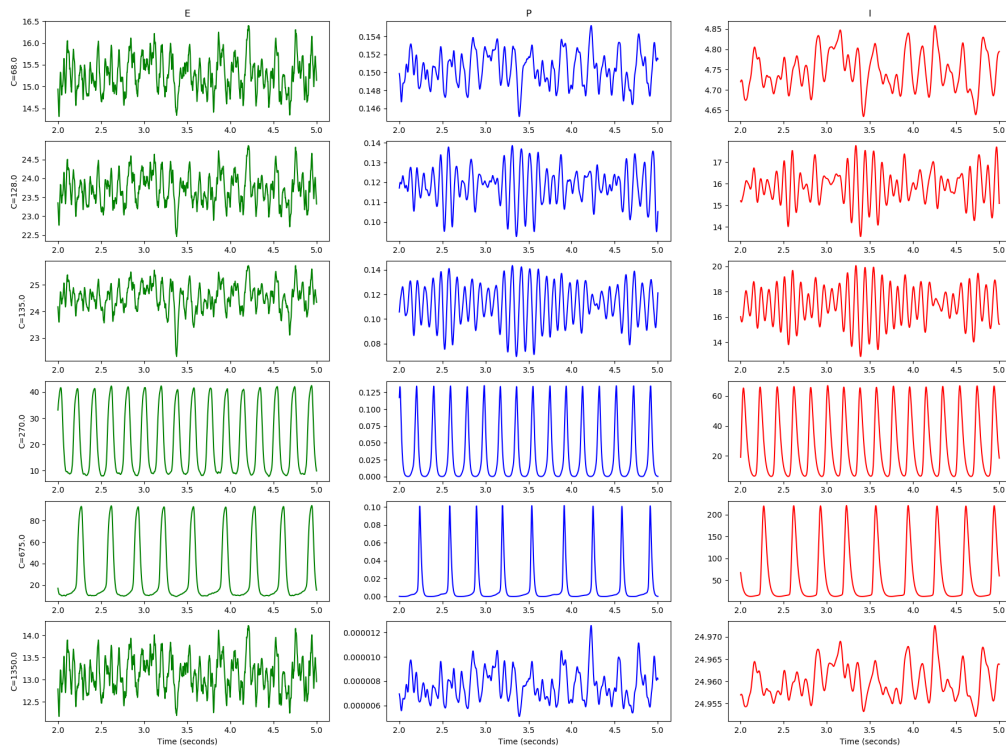
5.11.13 Example: Kremer_et_al_2011_barrel_cortex

Late Emergence of the Whisker Direction Selectivity Map in the Rat Barrel Cortex. Kremer Y, Leger JF, Goodman DF, Brette R, Bourdieu L (2011). J Neurosci 31(29):10689-700.

Development of direction maps with pinwheels in the barrel cortex. Whiskers are deflected with random moving bars.

N.B.: network construction can be long.

```python
from brian2 import *
import time

t1 = time.time()

# PARAMETERS
# Neuron numbers
M4, M23exc, M23inh = 22, 25, 12  # size of each barrel (in neurons)
N4, N23exc, N23inh = M4**2, M23exc**2, M23inh**2  # neurons per barrel
barrelarraysize = 5  # Choose 3 or 4 if memory error
Nbarrels = barrelarraysize**2
# Stimulation
stim_change_time = 5*ms
Fmax = .5/stim_change_time # maximum firing rate in layer 4 (.5 spike / stimulation)
# Neuron parameters
taum, taue, taui = 10*ms, 2*ms, 25*ms
El = -70*mV
Vt, vt_inc, tauvt = -55*mV, 2*mV, 50*ms  # adaptive threshold
# STDP
taup, taud = 5*ms, 25*ms
Ap, Ad= .05, -.04
# EPSPs/IPSPs
EPSP, IPSP = 1*mV, -1*mV
EPSC = EPSP * (taue/taum)**(taum/(taue-taum))
IPSC = IPSP * (taui/taum)**(taum/(taui-taum))
Ap, Ad = Ap*EPSC, Ad*EPSC

# Layer 4, models the input stimulus
eqs_layer4 = '''
rate = int(is_active)*clip(cos(direction - selectivity), 0, inf)*Fmax: Hz
is_active = abs((barrel_x + 0.5 - bar_x) * cos(direction) + (barrel_y + 0.5 - bar_y)␣
↪* sin(direction)) < 0.5: boolean
barrel_x : integer # The x index of the barrel
barrel_y : integer # The y index of the barrel
selectivity : 1
# Stimulus parameters (same for all neurons)
bar_x = cos(direction)*(t - stim_start_time)/(5*ms) + stim_start_x : 1 (shared)
bar_y = sin(direction)*(t - stim_start_time)/(5*ms) + stim_start_y : 1 (shared)
direction : 1 (shared) # direction of the current stimulus
stim_start_time : second (shared) # start time of the current stimulus
stim_start_x : 1 (shared) # start position of the stimulus
stim_start_y : 1 (shared) # start position of the stimulus
'''
layer4 = NeuronGroup(N4*Nbarrels, eqs_layer4, threshold='rand() < rate*dt',
                     method='euler', name='layer4')
layer4.barrel_x = '(i // N4) % barrelarraysize + 0.5'
layer4.barrel_y = 'i // (barrelarraysize*N4) + 0.5'
layer4.selectivity = '(i%N4)/(1.0*N4)*2*pi'  # for each barrel, selectivity between 0␣
↪and 2*pi
```

```
stimradius = (11+1)*.5

# Chose a new randomly oriented bar every 60ms
runner_code = '''
direction = rand()*2*pi
stim_start_x = barrelarraysize / 2.0 - cos(direction)*stimradius
stim_start_y = barrelarraysize / 2.0 - sin(direction)*stimradius
stim_start_time = t
'''
layer4.run_regularly(runner_code, dt=60*ms, when='start')

# Layer 2/3
# Model: IF with adaptive threshold
eqs_layer23 = '''
dv/dt=(ge+gi+El-v)/taum : volt
dge/dt=-ge/taue : volt
dgi/dt=-gi/taui : volt
dvt/dt=(Vt-vt)/tauvt : volt # adaptation
barrel_idx : integer
x : 1  # in "barrel width" units
y : 1  # in "barrel width" units
'''
layer23 = NeuronGroup(Nbarrels*(N23exc+N23inh), eqs_layer23,
                      threshold='v>vt', reset='v = El; vt += vt_inc',
                      refractory=2*ms, method='euler', name='layer23')
layer23.v = El
layer23.vt = Vt

# Subgroups for excitatory and inhibitory neurons in layer 2/3
layer23exc = layer23[:Nbarrels*N23exc]
layer23inh = layer23[Nbarrels*N23exc:]

# Layer 2/3 excitatory
# The units for x and y are the width/height of a single barrel
layer23exc.x = '(i % (barrelarraysize*M23exc)) * (1.0/M23exc)'
layer23exc.y = '(i // (barrelarraysize*M23exc)) * (1.0/M23exc)'
layer23exc.barrel_idx = 'floor(x) + floor(y)*barrelarraysize'

# Layer 2/3 inhibitory
layer23inh.x = 'i % (barrelarraysize*M23inh) * (1.0/M23inh)'
layer23inh.y = 'i // (barrelarraysize*M23inh) * (1.0/M23inh)'
layer23inh.barrel_idx = 'floor(x) + floor(y)*barrelarraysize'

print("Building synapses, please wait...")
# Feedforward connections (plastic)
feedforward = Synapses(layer4, layer23exc,
                       model='''w:volt
                                dA_source/dt = -A_source/taup : volt (event-driven)
                                dA_target/dt = -A_target/taud : volt (event-driven)''
↪',
                       on_pre='''ge+=w
                              A_source += Ap
                              w = clip(w+A_target, 0*volt, EPSC)''',
                       on_post='''
                              A_target += Ad
                              w = clip(w+A_source, 0*volt, EPSC)''',
```

```python
                           name='feedforward')
# Connect neurons in the same barrel with 50% probability
feedforward.connect('(barrel_x_pre + barrelarraysize*barrel_y_pre) == barrel_idx_post
↪',
                    p=0.5)
feedforward.w = EPSC*.5

print('excitatory lateral')
# Excitatory lateral connections
recurrent_exc = Synapses(layer23exc, layer23, model='w:volt', on_pre='ge+=w',
                         name='recurrent_exc')
recurrent_exc.connect(p='.15*exp(-.5*(((x_pre-x_post)/.4)**2+((y_pre-y_post)/.4)**2))
↪')
recurrent_exc.w['j<Nbarrels*N23exc'] = EPSC*.3 # excitatory->excitatory
recurrent_exc.w['j>=Nbarrels*N23exc'] = EPSC # excitatory->inhibitory


# Inhibitory lateral connections
print('inhibitory lateral')
recurrent_inh = Synapses(layer23inh, layer23exc, on_pre='gi+=IPSC',
                         name='recurrent_inh')
recurrent_inh.connect(p='exp(-.5*(((x_pre-x_post)/.2)**2+((y_pre-y_post)/.2)**2))')

if get_device().__class__.__name__=='RuntimeDevice':
    print('Total number of connections')
    print('feedforward: %d' % len(feedforward))
    print('recurrent exc: %d' % len(recurrent_exc))
    print('recurrent inh: %d' % len(recurrent_inh))

    t2 = time.time()
    print("Construction time: %.1fs" % (t2 - t1))

run(5*second, report='text')

# Calculate the preferred direction of each cell in layer23 by doing a
# vector average of the selectivity of the projecting layer4 cells, weighted
# by the synaptic weight.
_r = bincount(feedforward.j,
              weights=feedforward.w * cos(feedforward.selectivity_pre)/feedforward.N_
↪incoming,
              minlength=len(layer23exc))
_i = bincount(feedforward.j,
              weights=feedforward.w * sin(feedforward.selectivity_pre)/feedforward.N_
↪incoming,
              minlength=len(layer23exc))
selectivity_exc = (arctan2(_r, _i) % (2*pi))*180./pi


scatter(layer23.x[:Nbarrels*N23exc], layer23.y[:Nbarrels*N23exc],
        c=selectivity_exc[:Nbarrels*N23exc],
        edgecolors='none', marker='s', cmap='hsv')
vlines(np.arange(barrelarraysize), 0, barrelarraysize, 'k')
hlines(np.arange(barrelarraysize), 0, barrelarraysize, 'k')
clim(0, 360)
colorbar()
show()
```
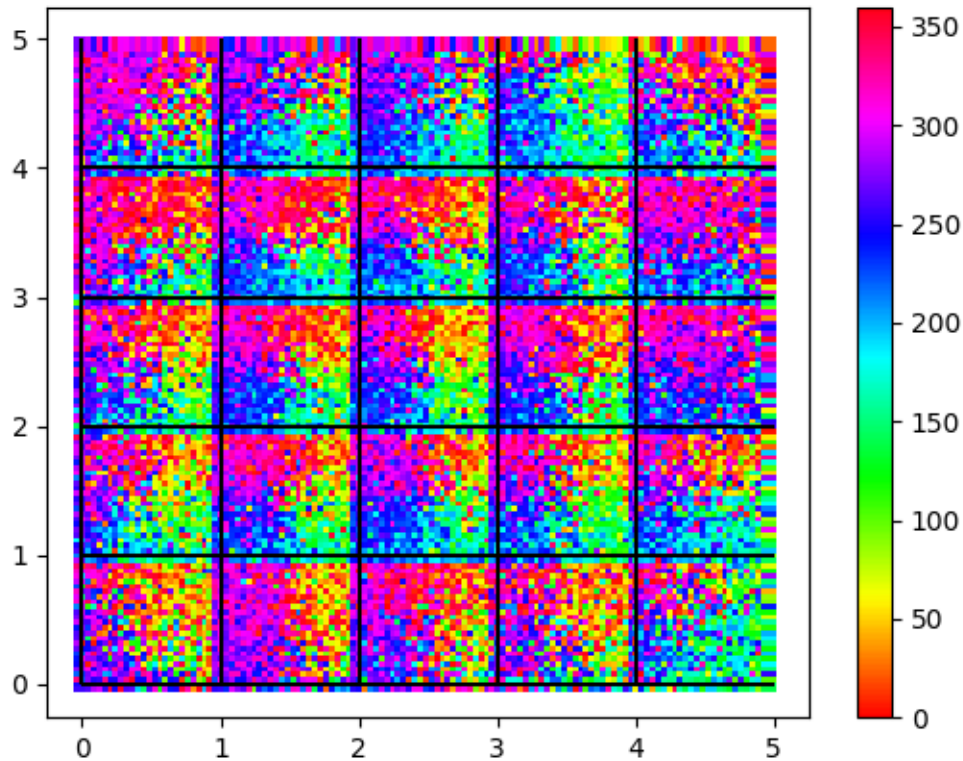
5.11.14 Example: Morris_Lecar_1981

Morris-Lecar model

Reproduces Fig. 9 of:

Catherine Morris and Harold Lecar. "Voltage Oscillations in the Barnacle Giant Muscle Fiber." Biophysical Journal 35, no. 1 (1981): 193–213.

```python
from brian2 import *
set_device('cpp_standalone')
defaultclock.dt = 0.01*ms

g_L = 2*mS
g_Ca = 4*mS
g_K = 8*mS
V_L = -50*mV
V_Ca = 100*mV
V_K = -70*mV
lambda_n__max = 1.0/(15*ms)
V_1 = 10*mV
V_2 = 15*mV   # Note that Figure caption says -15 which seems to be a typo
V_3 = -1*mV
```

```python
V_4 = 14.5*mV
C = 20*uF


# V,N-reduced system (Eq. 9 in article), note that the variables M and N (and lambda_
→N, etc.)
# have been renamed to m and n to better match the Hodgkin-Huxley convention, and
→because N has
# a reserved meaning in Brian (number of neurons)
eqs = '''
dV/dt = (-g_L*(V - V_L) - g_Ca*m_inf*(V - V_Ca) - g_K*n*(V - V_K) + I)/C : volt
dn/dt = lambda_n*(n_inf - n) : 1
m_inf = 0.5*(1 + tanh((V - V_1)/V_2)) : 1
n_inf = 0.5*(1 + tanh((V - V_3)/V_4)) : 1
lambda_n = lambda_n__max*cosh((V - V_3)/(2*V_4)) : Hz
I : amp
'''

neuron = NeuronGroup(17, eqs, method='exponential_euler')
neuron.I = (np.arange(17)*25+100)*uA
neuron.V = V_L
neuron.n = 'n_inf'
mon = StateMonitor(neuron, ['V', 'n'], record=True)

run_time = 220*ms
run(run_time)
fig, (ax1, ax2) = plt.subplots(1, 2, gridspec_kw={'right': 0.95, 'bottom': 0.15},
                               figsize=(6.4, 3.2))
fig.subplots_adjust(wspace=0.4)
for line_no, idx in enumerate([0, 4, 12, 15]):
    color = 'C%d' % line_no
    ax1.plot(mon.t/ms, mon.V[idx]/mV, color=color)
    ax1.text(225, mon.V[idx][-1]/mV, '%.0f' % (neuron.I[idx]/uA), color=color)
ax1.set(xlim=(0, 220), ylim=(-50, 50), xlabel='time (ms)')
ax1.set_ylabel('V (mV)', rotation=0)
ax1.spines['right'].set_visible(False)
ax1.spines['top'].set_visible(False)

# dV/dt nullclines
V = linspace(-50, 50, 100)*mV
for line_no, (idx, color) in enumerate([(0, 'C0'), (4, 'C1'), (8, 'C4'), (12, 'C2'),
→(16, 'C5')]):
    n_null = (g_L*(V - V_L) + g_Ca*0.5*(1 + tanh((V - V_1)/V_2))*(V - V_Ca) - neuron.
→I[idx])/(-g_K*(V - V_K))
    ax2.plot(V/mV, n_null, color=color)
    ax2.text(V[20+5*line_no]/mV, n_null[20+5*line_no]+0.01, '%.0f' % (neuron.I[idx]/
→uA), color=color)
# dn/dt nullcline
n_null = 0.5*(1 + tanh((V - V_3)/V_4))
ax2.plot(V/mV, n_null, color='k')
ax2.set(xlim=(-50, 50), ylim=(0, 1), xlabel='V (mV)')
ax2.set_ylabel('n', rotation=0)
ax2.spines['right'].set_visible(False)
ax2.spines['top'].set_visible(False)
plt.show()
```
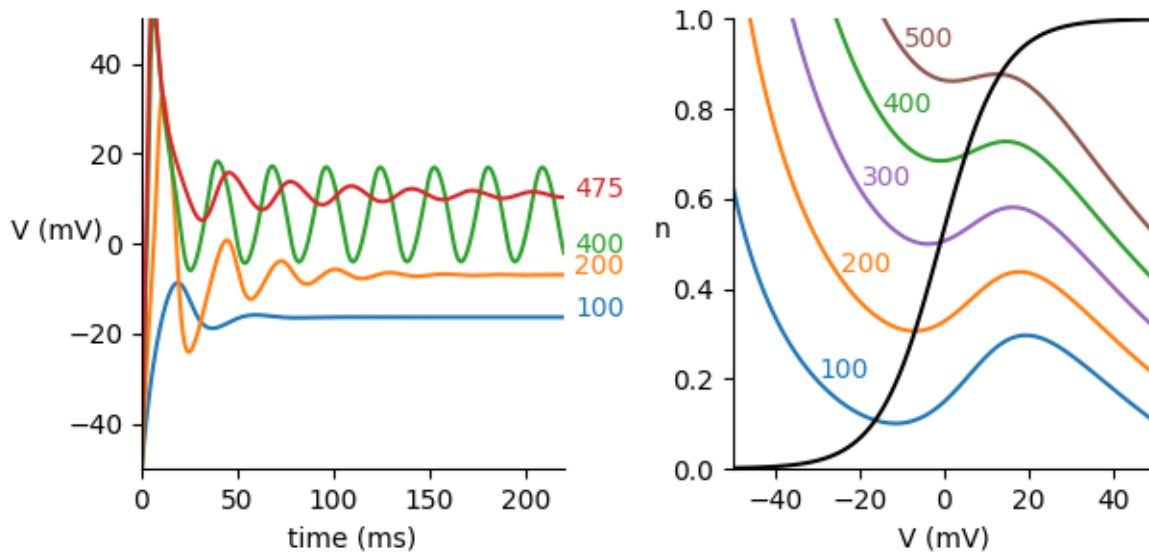
5.11.15 Example: Platkiewicz_Brette_2011

Slope-threshold relationship with noisy inputs, in the adaptive threshold model

Fig. 5E,F from:

Platkiewicz J and R Brette (2011). Impact of Fast Sodium Channel Inactivation on Spike Threshold Dynamics and Synaptic Integration. PLoS Comp Biol 7(5): e1001129. doi:10.1371/journal.pcbi.1001129

```python
from scipy import optimize
from scipy.stats import linregress

from brian2 import *

N = 200   # 200 neurons to get more statistics, only one is shown
duration = 1*second
# --Biophysical parameters
ENa = 60*mV
EL = -70*mV
vT = -55*mV
Vi = -63*mV
tauh = 5*ms
tau = 5*ms
ka = 5*mV
ki = 6*mV
a = ka / ki
tauI = 5*ms
mu = 15*mV
sigma = 6*mV / sqrt(tauI / (tauI + tau))

# --Theoretical prediction for the slope-threshold relationship (approximation:␣
↪a=1+epsilon)
```

```python
thresh = lambda slope, a: Vi - slope * tauh * log(1 + (Vi - vT / a) / (slope * tauh))
# -----Exact calculation of the slope-threshold relationship
# (note that optimize.fsolve does not work with units, we therefore let th be a
# unitless quantity, i.e. the value in volt).
thresh_ex = lambda s: optimize.fsolve(lambda th: (a*s*tauh*exp((Vi-th*volt)/(s*tauh))-
→th*volt*(1-a)-a*(s*tauh+Vi)+vT)/volt,
                                      thresh(s, a))*volt


eqs = """
dv/dt=(EL-v+mu+sigma*I)/tau : volt
dtheta/dt=(vT+a*clip(v-Vi, 0*mV, inf*mV)-theta)/tauh : volt
dI/dt=-I/tauI+(2/tauI)**.5*xi : 1 # Ornstein-Uhlenbeck
"""
neurons = NeuronGroup(N, eqs, threshold="v>theta", reset='v=EL',
                      refractory=5*ms)
neurons.v = EL
neurons.theta = vT
neurons.I = 0
S = SpikeMonitor(neurons)
M = StateMonitor(neurons, 'v', record=True)
Mt = StateMonitor(neurons, 'theta', record=0)


run(duration, report='text')

# Linear regression gives depolarization slope before spikes
tx = M.t[(M.t > 0*second) & (M.t < 1.5 * tauh)]
slope, threshold = [], []

for (i, t) in zip(S.i, S.t):
    ind = (M.t < t) & (M.t > t - tauh)
    mx = M.v[i, ind]
    s, _, _, _, _ = linregress(tx[:len(mx)]/ms, mx/mV)
    slope.append(s)
    threshold.append(mx[-1])

# Figure
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

ax1.plot(M.t/ms, M.v[0]/mV, 'k')
ax1.plot(Mt.t/ms, Mt.theta[0]/mV, 'r')
# Display spikes on the trace
spike_timesteps = np.round(S.t[S.i == 0]/defaultclock.dt).astype(int)
ax1.vlines(S.t[S.i == 0]/ms,
           M.v[0, spike_timesteps]/mV,
           0, color='r')
ax1.plot(S.t[S.i == 0]/ms, M.v[0, spike_timesteps]/mV, 'ro', ms=3)
ax1.set(xlabel='Time (ms)', ylabel='Voltage (mV)', xlim=(0, 500),
        ylim=(-75, -35))

ax2.plot(slope, Quantity(threshold)/mV, 'r.')
sx = linspace(0.5*mV/ms, 4*mV/ms, 100)
t = Quantity([thresh_ex(s) for s in sx])
ax2.plot(sx/(mV/ms), t/mV, 'k')
ax2.set(xlim=(0.5, 4), xlabel='Depolarization slope (mV/ms)',
        ylabel='Threshold (mV)')

fig.tight_layout()
```

```
plt.show()
```
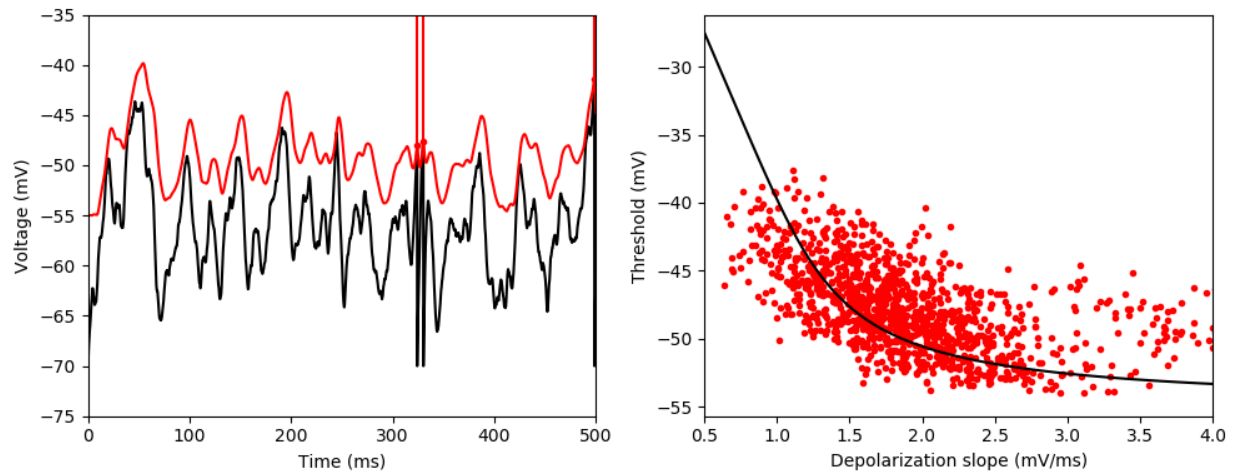

5.11.16 Example: Rossant_et_al_2011bis

5.11.17 Distributed synchrony example

Fig. 14 from:

> Rossant C, Leijon S, Magnusson AK, Brette R (2011). "Sensitivity of noisy neurons to coincident inputs".
> Journal of Neuroscience, 31(47).

5000 independent E/I Poisson inputs are injected into a leaky integrate-and-fire neuron. Synchronous events, following an independent Poisson process at 40 Hz, are considered, where 15 E Poisson spikes are randomly shifted to be synchronous at those events. The output firing rate is then significantly higher, showing that the spike timing of less than 1% of the excitatory synapses have an important impact on the postsynaptic firing.

```python
from brian2 import *

# neuron parameters
theta = -55*mV
El = -65*mV
vmean = -65*mV
taum = 5*ms
taue = 3*ms
taui = 10*ms
eqs = Equations("""
            dv/dt  = (ge+gi-(v-El))/taum : volt
            dge/dt = -ge/taue : volt
            dgi/dt = -gi/taui : volt
            """)

# input parameters
p = 15
ne = 4000
```

```
ni = 1000
lambdac = 40*Hz
lambdae = lambdai = 1*Hz

# synapse parameters
we = .5*mV/(taum/taue)**(taum/(taue-taum))
wi = (vmean-El-lambdae*ne*we*taue)/(lambdae*ni*taui)

# NeuronGroup definition
group = NeuronGroup(N=2, model=eqs, reset='v = El',
                    threshold='v>theta',
                    refractory=5*ms, method='exact')
group.v = El
group.ge = group.gi = 0

# independent E/I Poisson inputs
p1 = PoissonInput(group[0:1], 'ge', N=ne, rate=lambdae, weight=we)
p2 = PoissonInput(group[0:1], 'gi', N=ni, rate=lambdai, weight=wi)

# independent E/I Poisson inputs + synchronous E events
p3 = PoissonInput(group[1:], 'ge', N=ne, rate=lambdae-(p*1.0/ne)*lambdac, weight=we)
p4 = PoissonInput(group[1:], 'gi', N=ni, rate=lambdai, weight=wi)
p5 = PoissonInput(group[1:], 'ge', N=1, rate=lambdac, weight=p*we)

# run the simulation
M = SpikeMonitor(group)
SM = StateMonitor(group, 'v', record=True)
BrianLogger.log_level_info()
run(1*second)
# plot trace and spikes
for i in [0, 1]:
    spikes = (M.t[M.i == i] - defaultclock.dt)/ms
    val = SM[i].v
    subplot(2, 1, i+1)
    plot(SM.t/ms, val)
    plot(tile(spikes, (2, 1)),
         vstack((val[array(spikes, dtype=int)],
                 zeros(len(spikes)))), 'C0')
    title("%s: %d spikes/second" % (["uncorrelated inputs", "correlated inputs"][i],
                                    M.count[i]))
tight_layout()
show()
```

uncorrelated inputs: 2 spikes/second

correlated inputs: 9 spikes/second
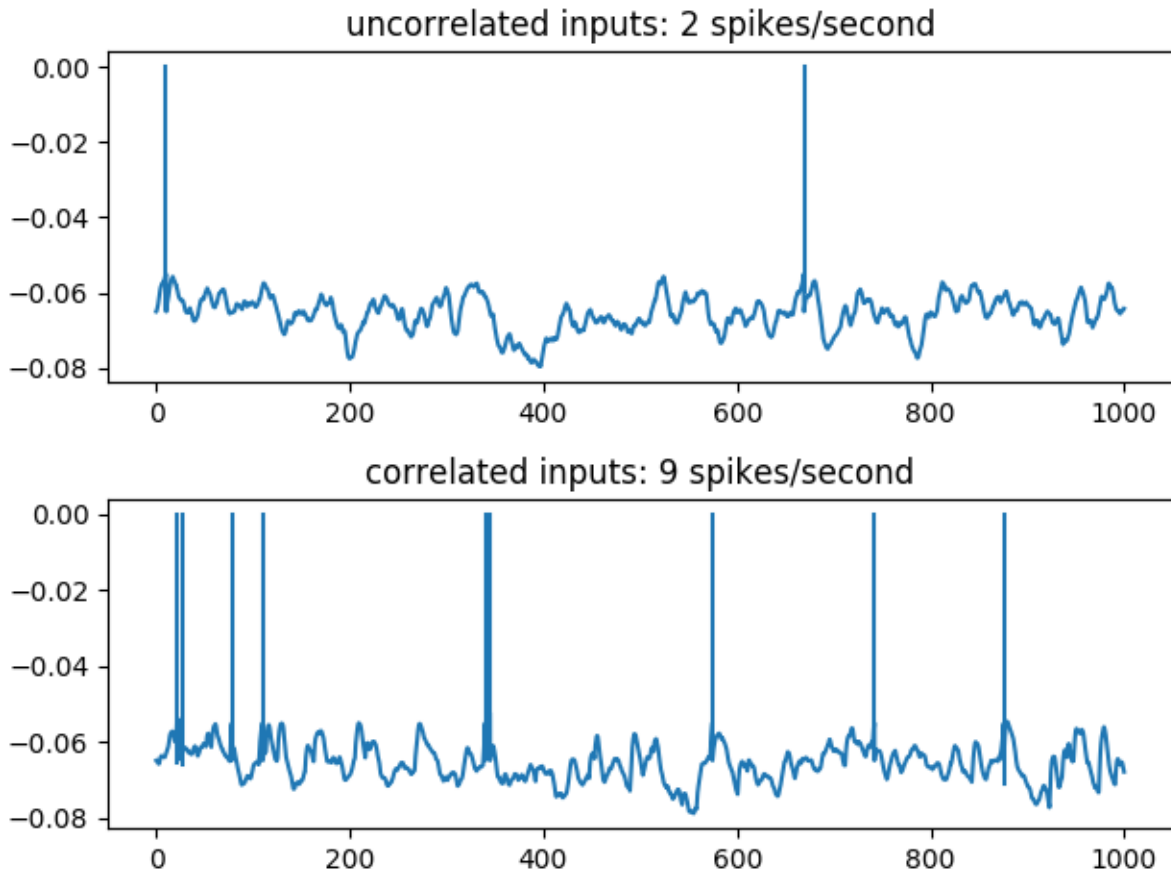
5.11.18 Example: Rothman_Manis_2003

Cochlear neuron model of Rothman & Manis

Rothman JS, Manis PB (2003) The roles potassium currents play in regulating the electrical activity of ventral cochlear nucleus neurons. J Neurophysiol 89:3097-113.

All model types differ only by the maximal conductances.

Adapted from their Neuron implementation by Romain Brette

```python
from brian2 import *

#defaultclock.dt=0.025*ms # for better precision

'''
Simulation parameters: choose current amplitude and neuron type
(from type1c, type1t, type12, type 21, type2, type2o)
'''
neuron_type = 'type1c'
Ipulse = 250*pA
```

(continues on next page)

```
C = 12*pF
Eh = -43*mV
EK = -70*mV   # -77*mV in mod file
El = -65*mV
ENa = 50*mV
nf = 0.85  # proportion of n vs p kinetics
zss = 0.5  # steady state inactivation of glt
temp = 22.  # temperature in degree celcius
q10 = 3. ** ((temp - 22) / 10.)
# hcno current (octopus cell)
frac = 0.0
qt = 4.5 ** ((temp - 33.) / 10.)

# Maximal conductances of different cell types in nS
maximal_conductances = dict(
type1c=(1000, 150, 0, 0, 0.5, 0, 2),
type1t=(1000, 80, 0, 65, 0.5, 0, 2),
type12=(1000, 150, 20, 0, 2, 0, 2),
type21=(1000, 150, 35, 0, 3.5, 0, 2),
type2=(1000, 150, 200, 0, 20, 0, 2),
type2o=(1000, 150, 600, 0, 0, 40, 2) # octopus cell
)
gnabar, gkhtbar, gkltbar, gkabar, ghbar, gbarno, gl = [x * nS for x in maximal_
↪conductances[neuron_type]]

# Classical Na channel
eqs_na = """
ina = gnabar*m**3*h*(ENa-v) : amp
dm/dt=q10*(minf-m)/mtau : 1
dh/dt=q10*(hinf-h)/htau : 1
minf = 1./(1+exp(-(vu + 38.) / 7.)) : 1
hinf = 1./(1+exp((vu + 65.) / 6.)) : 1
mtau =  ((10. / (5*exp((vu+60.) / 18.) + 36.*exp(-(vu+60.) / 25.))) + 0.04)*ms :␣
↪second
htau =  ((100. / (7*exp((vu+60.) / 11.) + 10.*exp(-(vu+60.) / 25.))) + 0.6)*ms :␣
↪second
"""

# KHT channel (delayed-rectifier K+)
eqs_kht = """
ikht = gkhtbar*(nf*n**2 + (1-nf)*p)*(EK-v) : amp
dn/dt=q10*(ninf-n)/ntau : 1
dp/dt=q10*(pinf-p)/ptau : 1
ninf =   (1 + exp(-(vu + 15) / 5.))**-0.5 : 1
pinf =  1. / (1 + exp(-(vu + 23) / 6.)) : 1
ntau =  ((100. / (11*exp((vu+60) / 24.) + 21*exp(-(vu+60) / 23.))) + 0.7)*ms : second
ptau = ((100. / (4*exp((vu+60) / 32.) + 5*exp(-(vu+60) / 22.))) + 5)*ms : second
"""

# Ih channel (subthreshold adaptive, non-inactivating)
eqs_ih = """
ih = ghbar*r*(Eh-v) : amp
dr/dt=q10*(rinf-r)/rtau : 1
rinf = 1. / (1+exp((vu + 76.) / 7.)) : 1
rtau = ((100000. / (237.*exp((vu+60.) / 12.) + 17.*exp(-(vu+60.) / 14.))) + 25.)*ms :␣
↪second
```

```python
"""

# KLT channel (low threshold K+)
eqs_klt = """
iklt = gkltbar*w**4*z*(EK-v) : amp
dw/dt=q10*(winf-w)/wtau : 1
dz/dt=q10*(zinf-z)/ztau : 1
winf = (1. / (1 + exp(-(vu + 48.) / 6.)))**0.25 : 1
zinf = zss + ((1.-zss) / (1 + exp((vu + 71.) / 10.))) : 1
wtau = ((100. / (6.*exp((vu+60.) / 6.) + 16.*exp(-(vu+60.) / 45.))) + 1.5)*ms : second
ztau = ((1000. / (exp((vu+60.) / 20.) + exp(-(vu+60.) / 8.))) + 50)*ms : second
"""

# Ka channel (transient K+)
eqs_ka = """
ika = gkabar*a**4*b*c*(EK-v): amp
da/dt=q10*(ainf-a)/atau : 1
db/dt=q10*(binf-b)/btau : 1
dc/dt=q10*(cinf-c)/ctau : 1
ainf = (1. / (1 + exp(-(vu + 31) / 6.)))**0.25 : 1
binf = 1. / (1 + exp((vu + 66) / 7.))**0.5 : 1
cinf = 1. / (1 + exp((vu + 66) / 7.))**0.5 : 1
atau =  ((100. / (7*exp((vu+60) / 14.) + 29*exp(-(vu+60) / 24.))) + 0.1)*ms : second
btau =  ((1000. / (14*exp((vu+60) / 27.) + 29*exp(-(vu+60) / 24.))) + 1)*ms : second
ctau = ((90. / (1 + exp((-66-vu) / 17.))) + 10)*ms : second
"""

# Leak
eqs_leak = """
ileak = gl*(El-v) : amp
"""

# h current for octopus cells
eqs_hcno = """
ihcno = gbarno*(h1*frac + h2*(1-frac))*(Eh-v) : amp
dh1/dt=(hinfno-h1)/tau1 : 1
dh2/dt=(hinfno-h2)/tau2 : 1
hinfno = 1./(1+exp((vu+66.)/7.)) : 1
tau1 = bet1/(qt*0.008*(1+alp1))*ms : second
tau2 = bet2/(qt*0.0029*(1+alp2))*ms : second
alp1 = exp(1e-3*3*(vu+50)*9.648e4/(8.315*(273.16+temp))) : 1
bet1 = exp(1e-3*3*0.3*(vu+50)*9.648e4/(8.315*(273.16+temp))) : 1
alp2 = exp(1e-3*3*(vu+84)*9.648e4/(8.315*(273.16+temp))) : 1
bet2 = exp(1e-3*3*0.6*(vu+84)*9.648e4/(8.315*(273.16+temp))) : 1
"""

eqs = """
dv/dt = (ileak + ina + ikht + iklt + ika + ih + ihcno + I)/C : volt
vu = v/mV : 1  # unitless v
I : amp
"""
eqs += eqs_leak + eqs_ka + eqs_na + eqs_ih + eqs_klt + eqs_kht + eqs_hcno

neuron = NeuronGroup(1, eqs, method='exponential_euler')
neuron.v = El

run(50*ms, report='text')  # Go to rest
```

```
M = StateMonitor(neuron, 'v', record=0)
neuron.I = Ipulse

run(100*ms, report='text')

plot(M.t / ms, M[0].v / mV)
xlabel('t (ms)')
ylabel('v (mV)')
show()
```
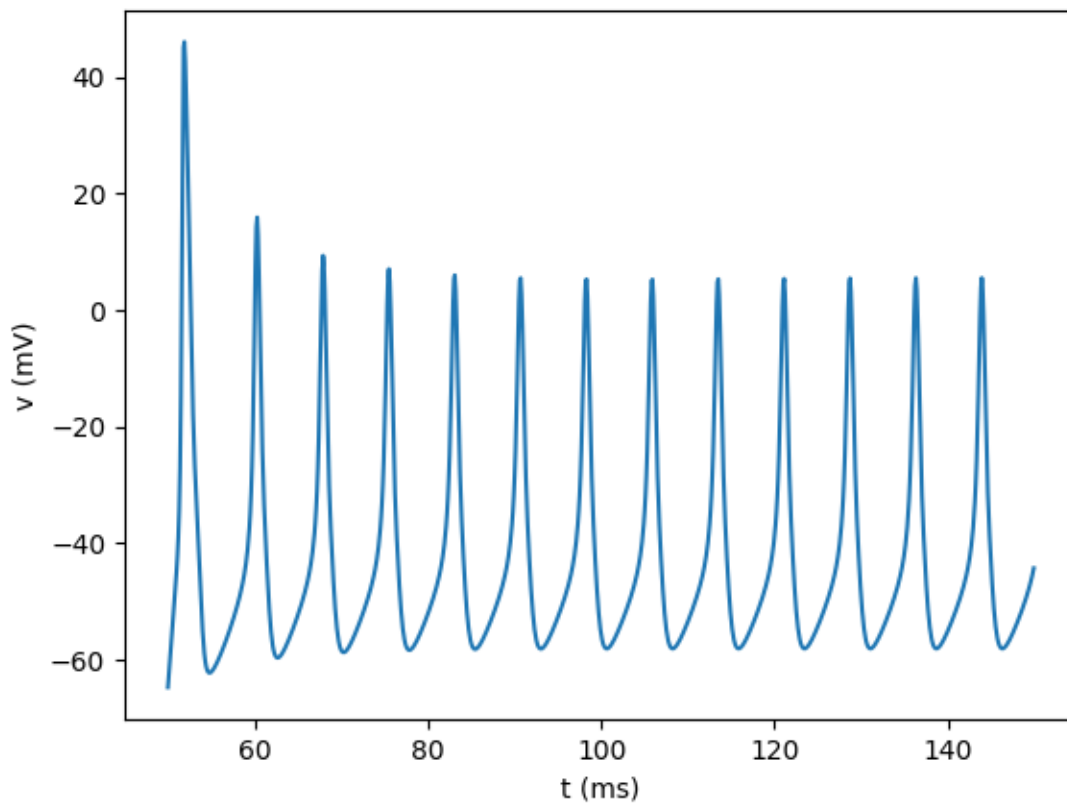

5.11.19 Example: Sturzl_et_al_2000

Adapted from Theory of Arachnid Prey Localization W. Sturzl, R. Kempter, and J. L. van Hemmen PRL 2000

Poisson inputs are replaced by integrate-and-fire neurons

Romain Brette

```
from brian2 import *

# Parameters
```

```python
degree = 2 * pi / 360.
duration = 500*ms
R = 2.5*cm  # radius of scorpion
vr = 50*meter/second  # Rayleigh wave speed
phi = 144*degree  # angle of prey
A = 250*Hz
deltaI = .7*ms  # inhibitory delay
gamma = (22.5 + 45 * arange(8)) * degree  # leg angle
delay = R / vr * (1 - cos(phi - gamma))  # wave delay

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

# Leg mechanical receptors
tau_legs = 1 * ms
sigma = .01
eqs_legs = """
dv/dt = (1 + rates(t - d) - v)/tau_legs + sigma*(2./tau_legs)**.5*xi:1
d : second
"""
legs = NeuronGroup(8, model=eqs_legs, threshold='v > 1', reset='v = 0',
                   refractory=1*ms, method='euler')
legs.d = delay
spikes_legs = SpikeMonitor(legs)

# Command neurons
tau = 1 * ms
taus = 1.001 * ms
wex = 7
winh = -2
eqs_neuron = '''
dv/dt = (x - v)/tau : 1
dx/dt = (y - x)/taus : 1 # alpha currents
dy/dt = -y/taus : 1
'''
neurons = NeuronGroup(8, model=eqs_neuron, threshold='v>1', reset='v=0',
                      method='exact')
synapses_ex = Synapses(legs, neurons, on_pre='y+=wex')
synapses_ex.connect(j='i')
synapses_inh = Synapses(legs, neurons, on_pre='y+=winh', delay=deltaI)
synapses_inh.connect('abs(((j - i) % N_post) - N_post/2) <= 1')
spikes = SpikeMonitor(neurons)

run(duration, report='text')

nspikes = spikes.count
```

```
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi/degree))
print("Estimated angle (deg): %.2f" % (phi_est/degree))
rmax = amax(nspikes)/duration/Hz
polar(concatenate((gamma, [gamma[0] + 2 * pi])),
      concatenate((nspikes, [nspikes[0]])) / duration / Hz,
      c='k')
axvline(phi, ls='-', c='g')
axvline(phi_est, ls='-', c='b')
show()
```
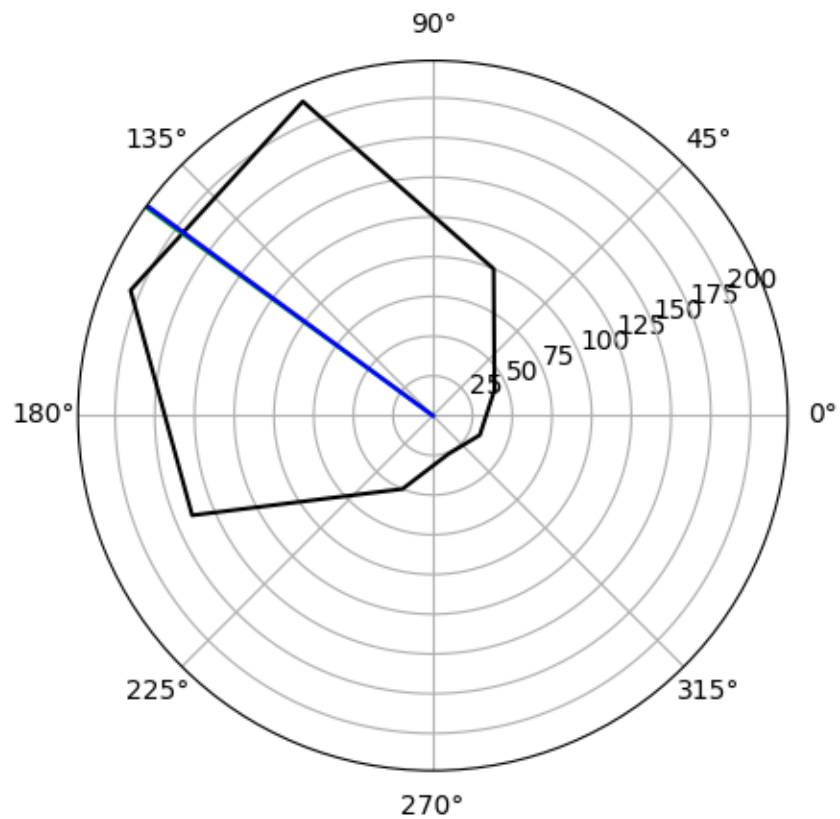
5.11.20 Example: Touboul_Brette_2008

Chaos in the AdEx model

Fig. 8B from: Touboul, J. and Brette, R. (2008). Dynamics and bifurcations of the adaptive exponential integrate-and-fire model. Biological Cybernetics 99(4-5):319-34.

This shows the bifurcation structure when the reset value is varied (vertical axis shows the values of w at spike times for a given a reset value Vr).

```python
from brian2 import *

defaultclock.dt = 0.01*ms

C = 281*pF
gL = 30*nS
EL = -70.6*mV
VT = -50.4*mV
DeltaT = 2*mV
tauw = 40*ms
a = 4*nS
b = 0.08*nA
I = .8*nA
Vcut = VT + 5 * DeltaT  # practical threshold condition
N = 200

eqs = """
dvm/dt=(gL*(EL-vm)+gL*DeltaT*exp((vm-VT)/DeltaT)+I-w)/C : volt
dw/dt=(a*(vm-EL)-w)/tauw : amp
Vr:volt
"""

neuron = NeuronGroup(N, model=eqs, threshold='vm > Vcut',
                     reset="vm = Vr; w += b", method='euler')
neuron.vm = EL
neuron.w = a * (neuron.vm - EL)
neuron.Vr = linspace(-48.3 * mV, -47.7 * mV, N)  # bifurcation parameter

init_time = 3*second
run(init_time, report='text')  # we discard the first spikes

states = StateMonitor(neuron, "w", record=True, when='start')
spikes = SpikeMonitor(neuron)
run(1 * second, report='text')

# Get the values of Vr and w for each spike
Vr = neuron.Vr[spikes.i]
w = states.w[spikes.i, int_((spikes.t-init_time)/defaultclock.dt)]

figure()
plot(Vr / mV, w / nA, '.k')
xlabel('Vr (mV)')
ylabel('w (nA)')
show()
```
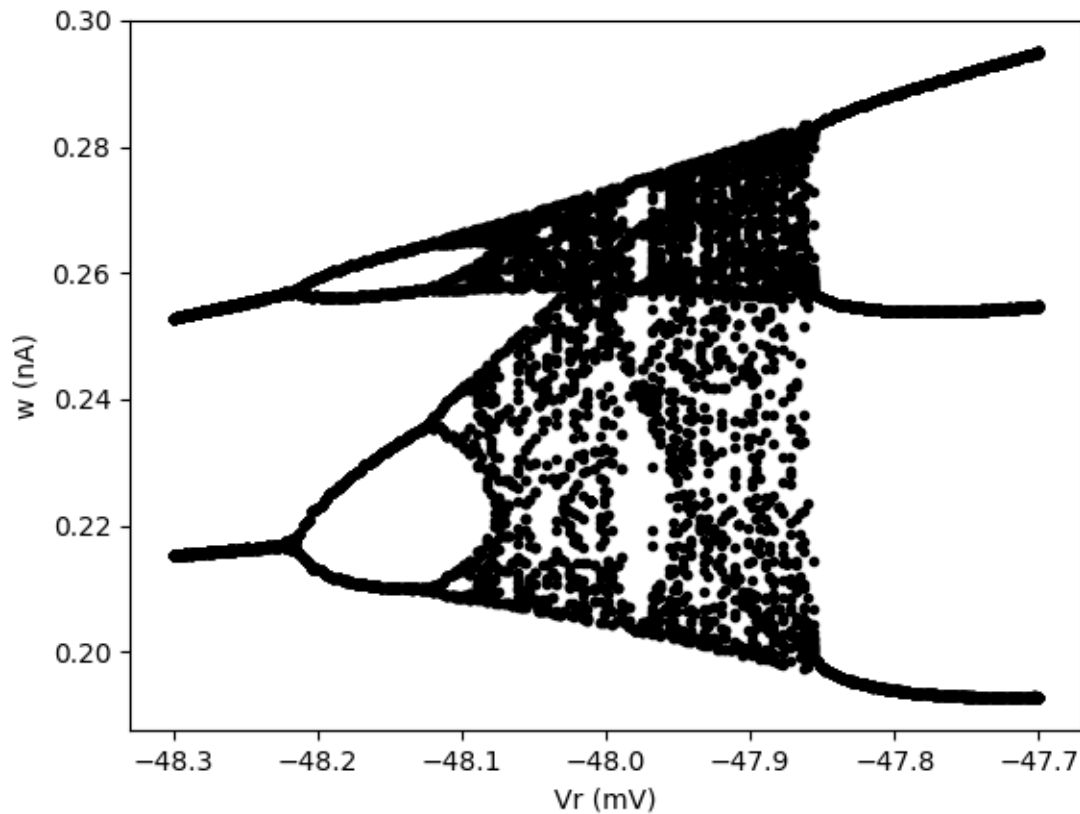
5.11.21 Example: Vogels_et_al_2011

Inhibitory synaptic plasticity in a recurrent network model

(F. Zenke, 2011) (from the 2012 Brian twister)

Adapted from:

Vogels, T. P., H. Sprekeler, F. Zenke, C. Clopath, and W. Gerstner. Inhibitory Plasticity Balances Excitation and Inhibition in Sensory Pathways and Memory Networks. Science (November 10, 2011).

```python
from brian2 import *

# #######################################
# Defining network model parameters
# #######################################

NE = 8000          # Number of excitatory cells
NI = NE/4          # Number of inhibitory cells

tau_ampa = 5.0*ms   # Glutamatergic synaptic time constant
```

```
tau_gaba = 10.0*ms  # GABAergic synaptic time constant
epsilon = 0.02      # Sparseness of synaptic connections

tau_stdp = 20*ms    # STDP time constant

simtime = 10*second # Simulation time

# ##########################################
# Neuron model
# ##########################################

gl = 10.0*nsiemens   # Leak conductance
el = -60*mV          # Resting potential
er = -80*mV          # Inhibitory reversal potential
vt = -50.*mV         # Spiking threshold
memc = 200.0*pfarad  # Membrane capacitance
bgcurrent = 200*pA   # External current

eqs_neurons='''
dv/dt=(-gl*(v-el)-(g_ampa*v+g_gaba*(v-er))+bgcurrent)/memc : volt (unless refractory)
dg_ampa/dt = -g_ampa/tau_ampa : siemens
dg_gaba/dt = -g_gaba/tau_gaba : siemens
'''

# ##########################################
# Initialize neuron group
# ##########################################

neurons = NeuronGroup(NE+NI, model=eqs_neurons, threshold='v > vt',
                      reset='v=el', refractory=5*ms, method='euler')
Pe = neurons[:NE]
Pi = neurons[NE:]

# ##########################################
# Connecting the network
# ##########################################

con_e = Synapses(Pe, neurons, on_pre='g_ampa += 0.3*nS')
con_e.connect(p=epsilon)
con_ii = Synapses(Pi, Pi, on_pre='g_gaba += 3*nS')
con_ii.connect(p=epsilon)

# ##########################################
# Inhibitory Plasticity
# ##########################################

eqs_stdp_inhib = '''
w : 1
dApre/dt=-Apre/tau_stdp : 1 (event-driven)
dApost/dt=-Apost/tau_stdp : 1 (event-driven)
'''
alpha = 3*Hz*tau_stdp*2  # Target rate parameter
gmax = 100               # Maximum inhibitory weight

con_ie = Synapses(Pi, Pe, model=eqs_stdp_inhib,
                  on_pre='''Apre += 1.
                     w = clip(w+(Apost-alpha)*eta, 0, gmax)
```

```
                          g_gaba += w*nS''',
                  on_post='''Apost += 1.
                          w = clip(w+Apre*eta, 0, gmax)
                  ''')
con_ie.connect(p=epsilon)
con_ie.w = 1e-10


# #########################################
# Setting up monitors
# #########################################

sm = SpikeMonitor(Pe)


# #########################################
# Run without plasticity
# #########################################
eta = 0           # Learning rate
run(1*second)


# #########################################
# Run with plasticity
# #########################################
eta = 1e-2          # Learning rate
run(simtime-1*second, report='text')


# #########################################
# Make plots
# #########################################

i, t = sm.it
subplot(211)
plot(t/ms, i, 'k.', ms=0.25)
title("Before")
xlabel("")
yticks([])
xlim(0.8*1e3, 1*1e3)
subplot(212)
plot(t/ms, i, 'k.', ms=0.25)
xlabel("time (ms)")
yticks([])
title("After")
xlim((simtime-0.2*second)/ms, simtime/ms)
show()
```
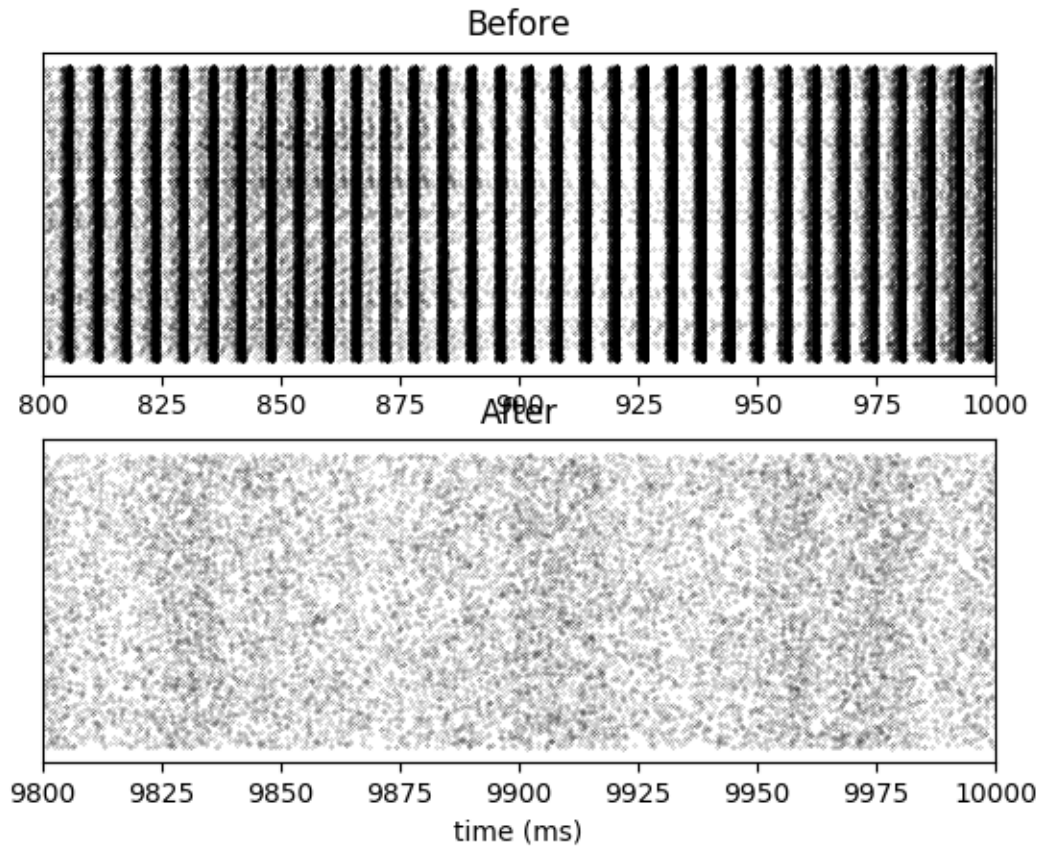
5.11.22 Example: Wang_Buszaki_1996

Wang-Buszaki model

J Neurosci. 1996 Oct 15;16(20):6402-13. Gamma oscillation by synaptic inhibition in a hippocampal interneuronal network model. Wang XJ, Buzsaki G.

Note that implicit integration (exponential Euler) cannot be used, and therefore simulation is rather slow.

```python
from brian2 import *

defaultclock.dt = 0.01*ms

Cm = 1*uF # /cm**2
Iapp = 2*uA
gL = 0.1*msiemens
EL = -65*mV
ENa = 55*mV
EK = -90*mV
gNa = 35*msiemens
gK = 9*msiemens
```

(continues on next page)

```
eqs = '''
dv/dt = (-gNa*m**3*h*(v-ENa)-gK*n**4*(v-EK)-gL*(v-EL)+Iapp)/Cm : volt
m = alpha_m/(alpha_m+beta_m) : 1
alpha_m = 0.1/mV*10*mV*exprel(-(v+35*mV)/(10*mV))/ms : Hz
beta_m = 4*exp(-(v+60*mV)/(18*mV))/ms : Hz
dh/dt = 5*(alpha_h*(1-h)-beta_h*h) : 1
alpha_h = 0.07*exp(-(v+58*mV)/(20*mV))/ms : Hz
beta_h = 1./(exp(-0.1/mV*(v+28*mV))+1)/ms : Hz
dn/dt = 5*(alpha_n*(1-n)-beta_n*n) : 1
alpha_n = 0.01/mV*10*mV*exprel(-(v+34*mV)/(10*mV))/ms : Hz
beta_n = 0.125*exp(-(v+44*mV)/(80*mV))/ms : Hz
'''

neuron = NeuronGroup(1, eqs, method='exponential_euler')
neuron.v = -70*mV
neuron.h = 1
M = StateMonitor(neuron, 'v', record=0)

run(100*ms, report='text')

plot(M.t/ms, M[0].v/mV)
show()
```
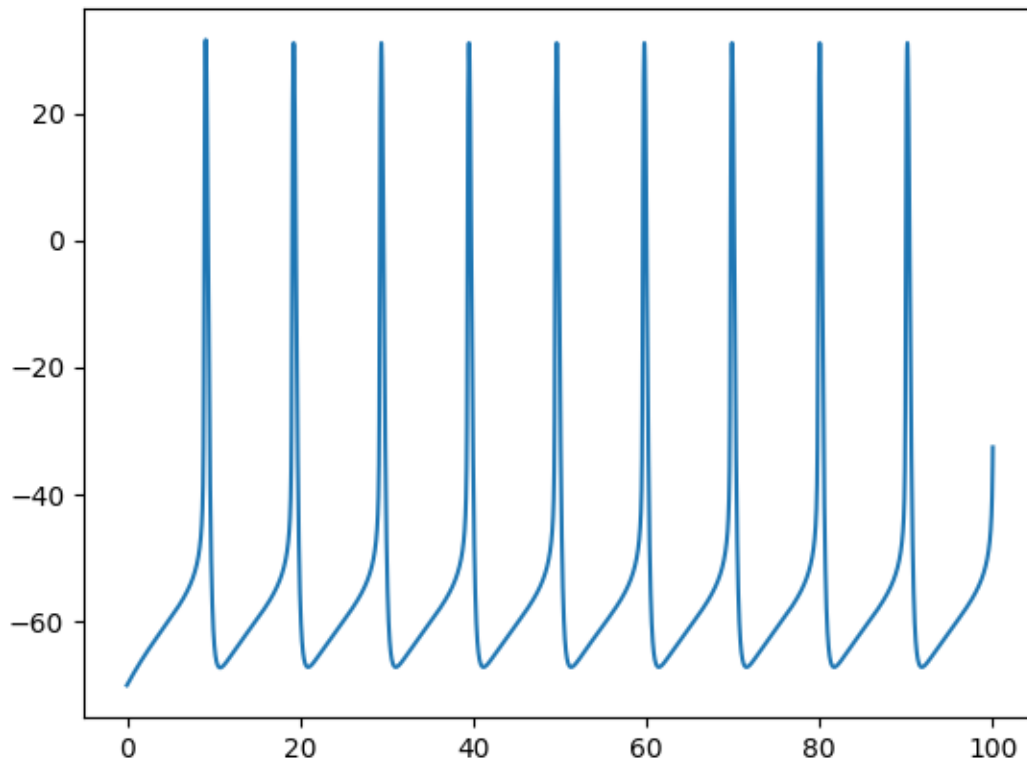
5.12 frompapers/Brette_2012

5.12.1 Example: Fig1

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig 1C-E. Somatic voltage-clamp in a ball-and-stick model with Na channels at a particular location.

```python
from brian2 import *
from params import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um)   # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

location = 40*um # where Na channels are placed
duration = 500*ms

# Channels
eqs='''
Im = gL*(EL - v) + gclamp*(vc - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum: 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gclamp : siemens/meter**2
gNa : siemens/meter**2
vc = EL + 50*mV * t/duration : volt (shared)  # Voltage clamp with a ramping voltage␣
↪command
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri)
compartment = morpho.axon[location]
neuron.v = EL
neuron.gclamp[0] = gL*500
neuron.gNa[compartment] = gNa_0/neuron.area[compartment]

# Monitors
mon = StateMonitor(neuron, ['v', 'vc', 'm'], record=True)

run(duration, report='text')

subplot(221)
plot(mon[0].vc/mV,
     -((mon[0].vc - mon[0].v)*(neuron.gclamp[0]))*neuron.area[0]/nA, 'k')
xlabel('V (mV)')
ylabel('I (nA)')
xlim(-75, -45)
title('I-V curve')

subplot(222)
plot(mon[0].vc/mV, mon[compartment].m, 'k')
xlabel('V (mV)')
ylabel('m')
```

(continues on next page)

```
title('Activation curve (m(V))')

subplot(223)
# Number of simulation time steps for each volt increment in the voltage-clamp
dt_per_volt = len(mon.t)/(50*mV)
for v in [-64*mV, -61*mV, -58*mV, -55*mV]:
    plot(mon.v[:100, int(dt_per_volt*(v - EL))]/mV, 'k')
xlabel('Distance from soma (um)')
ylabel('V (mV)')
title('Voltage across axon')

subplot(224)
plot(mon[compartment].v/mV, mon[compartment].v/mV, 'k--')   # Diagonal
plot(mon[0].v/mV, mon[compartment].v/mV, 'k')
xlabel('Vs (mV)')
ylabel('Va (mV)')
tight_layout()
show()
```
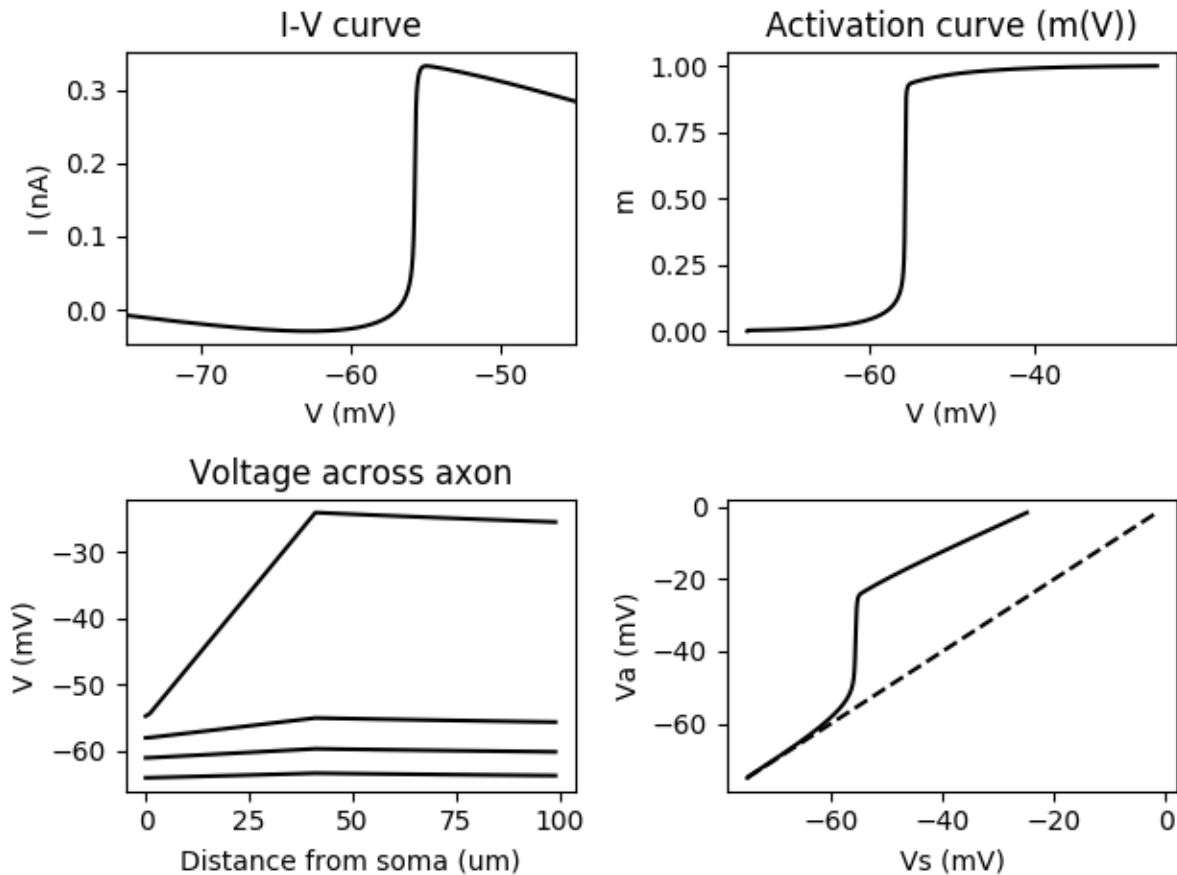
5.12.2 Example: Fig3AB

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 3. A, B. Kink with only Nav1.6 channels

```python
from brian2 import *
from params import *

codegen.target='numpy'

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um)   # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

location = 40*um   # where Na channels are placed

# Channels
eqs='''
Im = gL*(EL - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum : 1 # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gNa : siemens/meter**2
Iin : amp (point current)
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")

compartment = morpho.axon[location]
neuron.v = EL
neuron.gNa[compartment] = gNa_0/neuron.area[compartment]
M = StateMonitor(neuron, ['v', 'm'], record=True)

run(20*ms, report='text')
neuron.Iin[0] = gL * 20*mV * neuron.area[0]
run(80*ms, report='text')

subplot(121)
plot(M.t/ms, M[0].v/mV, 'r')
plot(M.t/ms, M[compartment].v/mV, 'k')
plot(M.t/ms, M[compartment].m*(80+60)-80, 'k--')   # open channels
ylim(-80, 60)
xlabel('Time (ms)')
ylabel('V (mV)')
title('Voltage traces')

subplot(122)
dm = diff(M[0].v) / defaultclock.dt
dm40 = diff(M[compartment].v) / defaultclock.dt
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment].v/mV)[1:], dm40/(volt/second), 'k')
xlim(-80, 40)
```

```
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot')
tight_layout()
show()
```
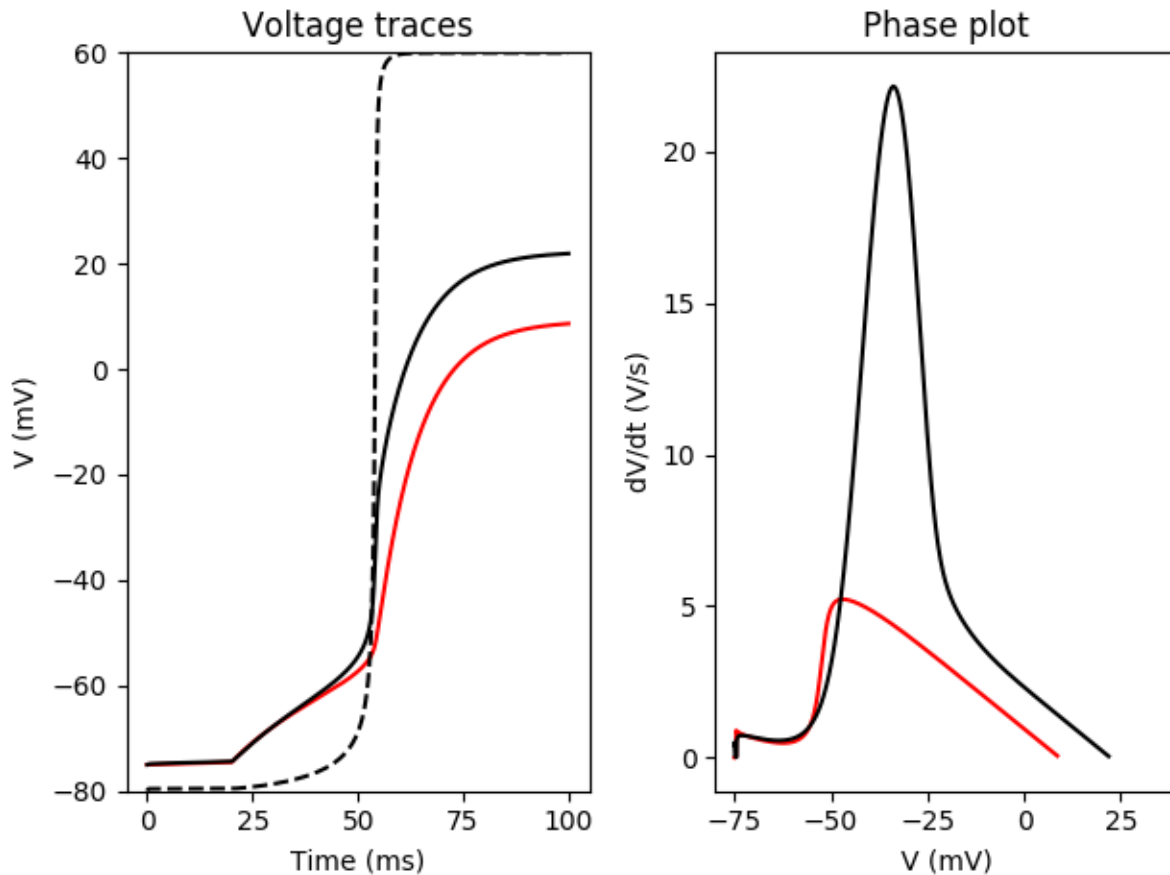

5.12.3 Example: Fig3CF

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 3C-F. Kink with Nav1.6 and Nav1.2

```
from brian2 import *
from params import *

defaultclock.dt = 0.01*ms

# Morphology
morpho = Soma(50*um) # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)
```

```python
location16 = 40*um  # where Nav1.6 channels are placed
location12 = 15*um  # where Nav1.2 channels are placed


va2 = va + 15*mV  # depolarized Nav1.2

# Channels
duration = 100*ms
eqs='''
Im = gL * (EL - v) + gNa*m*(ENa - v) + gNa2*m2*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum : 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
dm2/dt = (minf2 - m2) / taum : 1 # simplified Na channel, Nav1.2
minf2 = 1/(1 + exp((va2 - v) / ka)) : 1
gNa : siemens/meter**2
gNa2 : siemens/meter**2  # Nav1.2
Iin : amp (point current)
'''


neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")
compartment16 = morpho.axon[location16]
compartment12 = morpho.axon[location12]
neuron.v = EL
neuron.gNa[compartment16] = gNa_0/neuron.area[compartment16]
neuron.gNa2[compartment12] = 20*gNa_0/neuron.area[compartment12]
# Monitors
M = StateMonitor(neuron, ['v', 'm', 'm2'], record=True)

run(20*ms, report='text')
neuron.Iin[0] = gL * 20*mV * neuron.area[0]
run(80*ms, report='text')

subplot(221)
plot(M.t/ms, M[0].v/mV, 'r')
plot(M.t/ms, M[compartment16].v/mV, 'k')
plot(M.t/ms, M[compartment16].m*(80+60)-80, 'k--')  # open channels
ylim(-80, 60)
xlabel('Time (ms)')
ylabel('V (mV)')
title('Voltage traces')

subplot(222)
plot(M[0].v/mV, M[compartment16].m, 'k')
plot(M[0].v/mV, 1 / (1 + exp((va - M[0].v) / ka)), 'k--')
plot(M[0].v/mV, M[compartment12].m2, 'r')
plot(M[0].v/mV, 1 / (1 + exp((va2 - M[0].v) / ka)), 'r--')
xlim(-70, 0)
xlabel('V (mV)')
ylabel('m')
title('Activation curves')

subplot(223)
dm = diff(M[0].v) / defaultclock.dt
dm40 = diff(M[compartment16].v) / defaultclock.dt
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment16].v/mV)[1:], dm40/(volt/second), 'k')
```

```
xlim(-80, 40)
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot')

subplot(224)
plot((M[0].v/mV)[1:], dm/(volt/second), 'r')
plot((M[compartment16].v/mV)[1:], dm40/(volt/second), 'k')
plot((M[0].v/mV)[1:], 10 + 0*dm/(volt/second), 'k--')
xlim(-70, -40)
ylim(0, 20)
xlabel('V (mV)')
ylabel('dV/dt (V/s)')
title('Phase plot(zoom)')
tight_layout()
show()
```
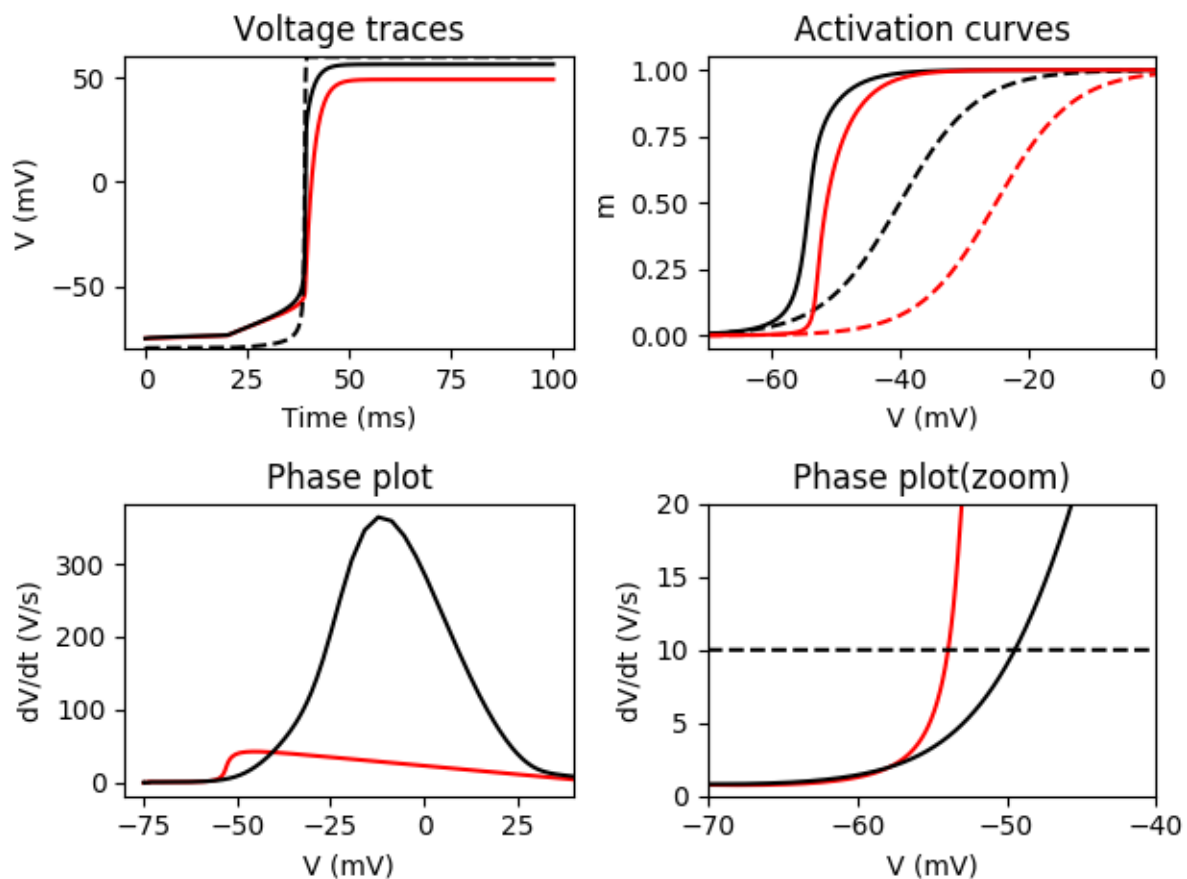
5.12.4 Example: Fig4

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 4E-F. Spatial distribution of Na channels. Tapering axon near soma.

```python
from brian2 import *
from params import *

defaultclock.dt = 0.025*ms

# Morphology
morpho = Soma(50*um) # chosen for a target Rm
# Tapering (change this for the other figure panels)
diameters = hstack([linspace(4, 1, 11), ones(290)])*um
morpho.axon = Section(diameter=diameters, length=ones(300)*um, n=300)

# Na channels
Na_start = (25 + 10)*um
Na_end = (40 + 10)*um
linear_distribution = True  # True is F, False is E

duration = 500*ms

# Channels
eqs='''
Im = gL*(EL - v) + gclamp*(vc - v) + gNa*m*(ENa - v) : amp/meter**2
dm/dt = (minf - m) / taum: 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gclamp : siemens/meter**2
gNa : siemens/meter**2
vc = EL + 50*mV * t / duration : volt (shared)  # Voltage clamp with a ramping↩
↪voltage command
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       method="exponential_euler")
compartments = morpho.axon[Na_start:Na_end]
neuron.v = EL
neuron.gclamp[0] = gL*500

if linear_distribution:
    profile = linspace(1, 0, len(compartments))
else:
    profile = ones(len(compartments))
profile = profile / sum(profile)  # normalization

neuron.gNa[compartments] = gNa_0 * profile / neuron.area[compartments]

# Monitors
mon = StateMonitor(neuron, 'v', record=True)

run(duration, report='text')

dt_per_volt = len(mon.t) / (50*mV)
```

(continues on next page)

```
for v in [-64*mV, -61*mV, -58*mV, -55*mV, -52*mV]:
    plot(mon.v[:100, int(dt_per_volt * (v - EL))]/mV, 'k')
xlim(0, 50+10)
ylim(-65, -25)
ylabel('V (mV)')
xlabel('Location (um)')
title('Voltage across axon')
tight_layout()
show()
```
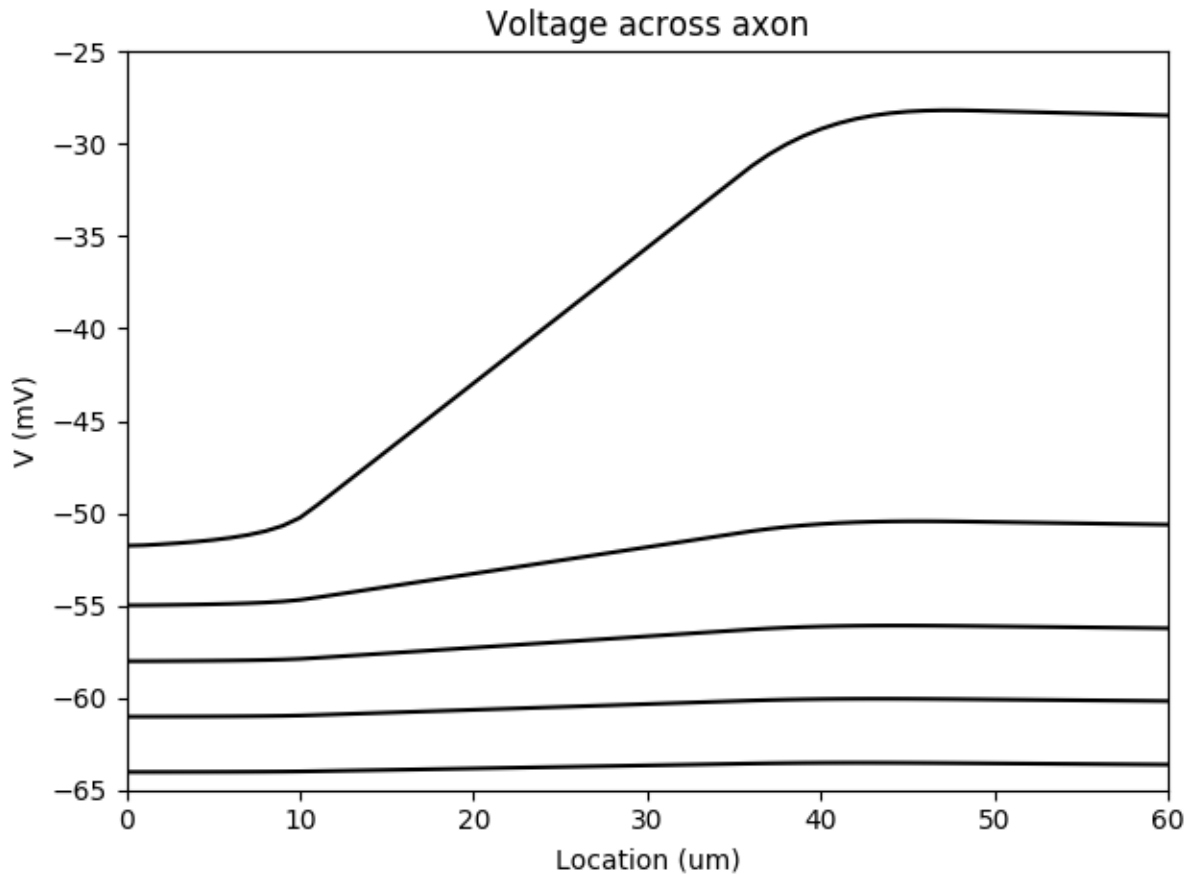

5.12.5 Example: Fig5A

Brette R (2013). Sharpness of spike initiation in neurons explained by compartmentalization. PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.

Fig. 5A. Voltage trace for current injection, with an additional reset when a spike is produced.

Trick: to reset the entire neuron, we use a set of synapses from the spike initiation compartment where the threshold condition applies to all compartments, and the reset operation (v = EL) is applied there every time a spike is produced.

```
from brian2 import *
from params import *
```

```python
defaultclock.dt = 0.025*ms
duration = 500*ms

# Morphology
morpho = Soma(50*um)  # chosen for a target Rm
morpho.axon = Cylinder(diameter=1*um, length=300*um, n=300)

# Input
taux = 5*ms
sigmax = 12*mV
xx0 = 7*mV


compartment = 40

# Channels
eqs = '''
Im = gL * (EL - v) + gNa * m * (ENa - v) + gLx * (xx0 + xx) : amp/meter**2
dm/dt = (minf - m) / taum : 1  # simplified Na channel
minf = 1 / (1 + exp((va - v) / ka)) : 1
gNa : siemens/meter**2
gLx : siemens/meter**2
dxx/dt = -xx / taux + sigmax * (2 / taux)**.5 *xi : volt
'''

neuron = SpatialNeuron(morphology=morpho, model=eqs, Cm=Cm, Ri=Ri,
                       threshold='m>0.5', threshold_location=compartment,
                       refractory=5*ms)
neuron.v = EL
neuron.gLx[0] = gL
neuron.gNa[compartment] = gNa_0 / neuron.area[compartment]

# Reset the entire neuron when there is a spike
reset = Synapses(neuron, neuron, on_pre='v = EL')
reset.connect('i == compartment')  # Connects the spike initiation compartment to all
↪compartments

# Monitors
S = SpikeMonitor(neuron)
M = StateMonitor(neuron, 'v', record=0)
run(duration, report='text')

# Add spikes for display
v = M[0].v
for t in S.t:
    v[int(t / defaultclock.dt)] = 50*mV

plot(M.t/ms, v/mV, 'k')
tight_layout()
show()
```
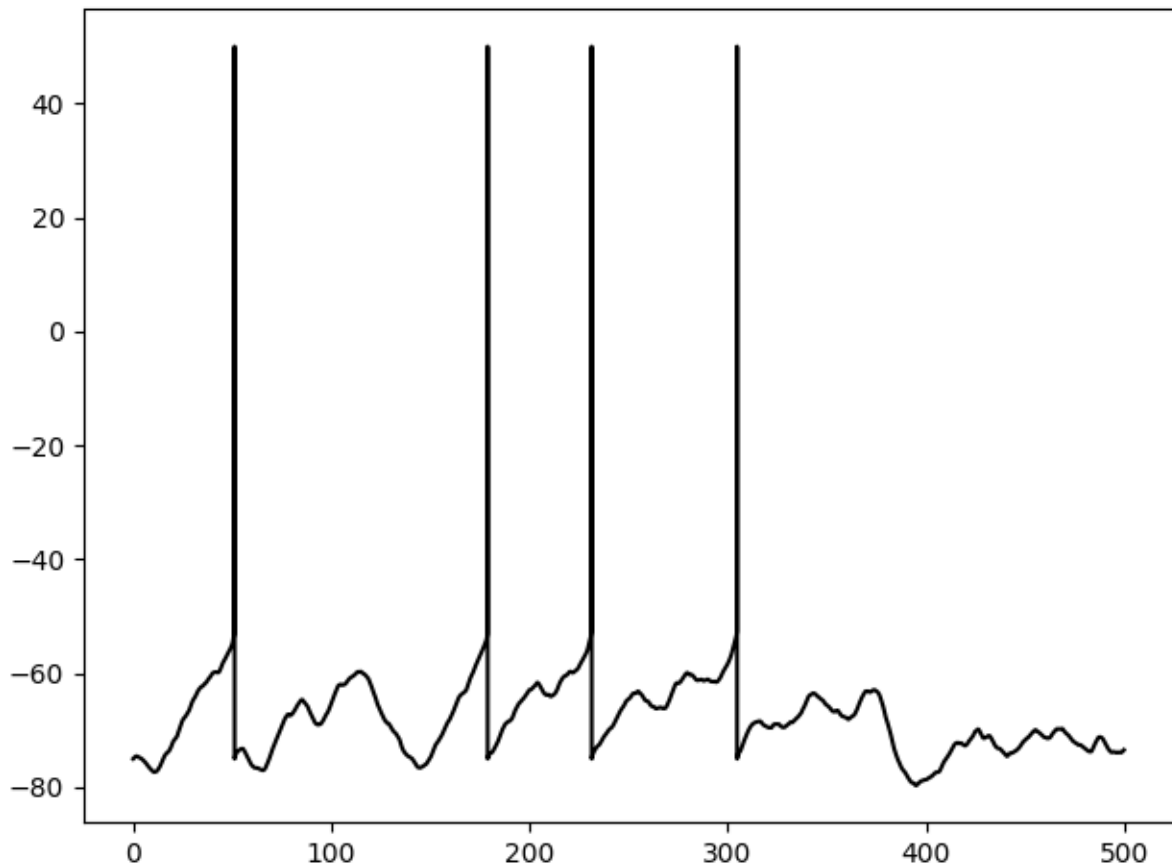
5.12.6 Example: params

Parameters for spike initiation simulations.

```
from brian2.units import *

# Passive parameters
EL = -75*mV
S = 7.85e-9*meter**2   # area (sphere of 50 um diameter)
Cm = 0.75*uF/cm**2
gL = 1. / (30000*ohm*cm**2)
Ri = 150*ohm*cm

# Na channels
ENa = 60*mV
ka = 6*mV
va = -40*mV
gNa_0 = gL * 2*S
taum = 0.1*ms
```

5.12.7 README.txt

```
These are Brian scripts corresponding to the following paper:

Brette R (2013). Sharpness of spike initiation in neurons explained by␣
↪compartmentalization.
PLoS Comp Biol, doi: 10.1371/journal.pcbi.1003338.


params.py    contains model parameters

Essential figures from the paper:
Fig1.py
Fig3AB.py
Fig3CD.py
Fig4.py
Fig5A.py
```

5.13 frompapers/Stimberg_et_al_2018

5.13.1 Example: example_1_COBA

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 1: Modeling of neurons and synapses.

Randomly connected networks with conductance-based synapses (COBA; see Brunel, 2000). Synapses exhibit short-time plasticity (Tsodyks, 2005; Tsodyks et al., 1998).

```python
from brian2 import *
import sympy

import plot_utils as pu

seed(11922)  # to get identical figures for repeated runs


################################################################################
# Model parameters
################################################################################
### General parameters
duration = 1.0*second   # Total simulation time
sim_dt = 0.1*ms         # Integrator/sampling step
N_e = 3200              # Number of excitatory neurons
N_i = 800               # Number of inhibitory neurons


### Neuron parameters
E_l = -60*mV            # Leak reversal potential
g_l = 9.99*nS           # Leak conductance
E_e = 0*mV              # Excitatory synaptic reversal potential
E_i = -80*mV            # Inhibitory synaptic reversal potential
C_m = 198*pF            # Membrane capacitance
tau_e = 5*ms            # Excitatory synaptic time constant
tau_i = 10*ms           # Inhibitory synaptic time constant
```

(continues on next page)

```
tau_r = 5*ms              # Refractory period
I_ex = 150*pA             # External current
V_th = -50*mV             # Firing threshold
V_r = E_l                 # Reset potential

### Synapse parameters
w_e = 0.05*nS             # Excitatory synaptic conductance
w_i = 1.0*nS              # Inhibitory synaptic conductance
U_0 = 0.6                 # Synaptic release probability at rest
Omega_d = 2.0/second      # Synaptic depression rate
Omega_f = 3.33/second   # Synaptic facilitation rate


#############################################################################
# Model definition
#############################################################################
# Set the integration time (in this case not strictly necessary, since we are
# using the default value)
defaultclock.dt = sim_dt

### Neurons
neuron_eqs = '''
dv/dt = (g_l*(E_l-v) + g_e*(E_e-v) + g_i*(E_i-v) +
        I_ex)/C_m    : volt (unless refractory)
dg_e/dt = -g_e/tau_e  : siemens  # post-synaptic exc. conductance
dg_i/dt = -g_i/tau_i  : siemens  # post-synaptic inh. conductance
'''
neurons = NeuronGroup(N_e + N_i, model=neuron_eqs,
                      threshold='v>V_th', reset='v=V_r',
                      refractory='tau_r', method='euler')
# Random initial membrane potential values and conductances
neurons.v = 'E_l + rand()*(V_th-E_l)'
neurons.g_e = 'rand()*w_e'
neurons.g_i = 'rand()*w_i'
exc_neurons = neurons[:N_e]
inh_neurons = neurons[N_e:]

### Synapses
synapses_eqs = '''
# Usage of releasable neurotransmitter per single action potential:
du_S/dt = -Omega_f * u_S     : 1 (event-driven)
# Fraction of synaptic neurotransmitter resources available:
dx_S/dt = Omega_d *(1 - x_S) : 1 (event-driven)
'''
synapses_action = '''
u_S += U_0 * (1 - u_S)
r_S = u_S * x_S
x_S -= r_S
'''
exc_syn = Synapses(exc_neurons, neurons, model=synapses_eqs,
                   on_pre=synapses_action+'g_e_post += w_e*r_S')
inh_syn = Synapses(inh_neurons, neurons, model=synapses_eqs,
                   on_pre=synapses_action+'g_i_post += w_i*r_S')

exc_syn.connect(p=0.05)
inh_syn.connect(p=0.2)
# Start from "resting" condition: all synapses have fully-replenished
# neurotransmitter resources
```

```
exc_syn.x_S = 1
inh_syn.x_S = 1


# ############################################################################
# # Monitors
# ############################################################################
# Note that we could use a single monitor for all neurons instead, but in this
# way plotting is a bit easier in the end
exc_mon = SpikeMonitor(exc_neurons)
inh_mon = SpikeMonitor(inh_neurons)


### We record some additional data from a single excitatory neuron
ni = 50
# Record conductances and membrane potential of neuron ni
state_mon = StateMonitor(exc_neurons, ['v', 'g_e', 'g_i'], record=ni)
# We make sure to monitor synaptic variables after synapse are updated in order
# to use simple recurrence relations to reconstruct them. Record all synapses
# originating from neuron ni
synapse_mon = StateMonitor(exc_syn, ['u_S', 'x_S'],
                           record=exc_syn[ni, :], when='after_synapses')


# ############################################################################
# # Simulation run
# ############################################################################
run(duration, report='text')


# ############################################################################
# Analysis and plotting
# ############################################################################
plt.style.use('figures.mplstyle')

### Spiking activity (w/ rate)
fig1, ax = plt.subplots(nrows=2, ncols=1, sharex=False,
                        gridspec_kw={'height_ratios': [3, 1],
                                     'left': 0.18, 'bottom': 0.18, 'top': 0.95,
                                     'hspace': 0.1},
                        figsize=(3.07, 3.07))
ax[0].plot(exc_mon.t[exc_mon.i <= N_e//4]/ms,
           exc_mon.i[exc_mon.i <= N_e//4], '|', color='C0')
ax[0].plot(inh_mon.t[inh_mon.i <= N_i//4]/ms,
           inh_mon.i[inh_mon.i <= N_i//4]+N_e//4, '|', color='C1')
pu.adjust_spines(ax[0], ['left'])
ax[0].set(xlim=(0., duration/ms), ylim=(0, (N_e+N_i)//4), ylabel='neuron index')

# Generate frequencies
bin_size = 1*ms
spk_count, bin_edges = np.histogram(np.r_[exc_mon.t/ms, inh_mon.t/ms],
                                    int(duration/ms))
rate = double(spk_count)/(N_e + N_i)/bin_size/Hz
ax[1].plot(bin_edges[:-1], rate, '-', color='k')
pu.adjust_spines(ax[1], ['left', 'bottom'])
ax[1].set(xlim=(0., duration/ms), ylim=(0, 10.),
          xlabel='time (ms)', ylabel='rate (Hz)')
pu.adjust_ylabels(ax, x_offset=-0.18)

### Dynamics of a single neuron
fig2, ax = plt.subplots(4, sharex=False,
```

```python
                            gridspec_kw={'left': 0.27, 'bottom': 0.18, 'top': 0.95,
                                         'hspace': 0.2},
                            figsize=(3.07, 3.07))
### Postsynaptic conductances
ax[0].plot(state_mon.t/ms, state_mon.g_e[0]/nS, color='C0')
ax[0].plot(state_mon.t/ms, -state_mon.g_i[0]/nS, color='C1')
ax[0].plot([state_mon.t[0]/ms, state_mon.t[-1]/ms], [0, 0], color='grey',
           linestyle=':')
# Adjust axis
pu.adjust_spines(ax[0], ['left'])
ax[0].set(xlim=(0., duration/ms), ylim=(-5.0, 0.25),
          ylabel=f"postsyn.\nconduct.\n(${sympy.latex(nS)}$)")


### Membrane potential
ax[1].axhline(V_th/mV, color='C2', linestyle=':')  # Threshold
# Artificially insert spikes
ax[1].plot(state_mon.t/ms, state_mon.v[0]/mV, color='black')
ax[1].vlines(exc_mon.t[exc_mon.i == ni]/ms, V_th/mV, 0, color='black')
pu.adjust_spines(ax[1], ['left'])
ax[1].set(xlim=(0., duration/ms), ylim=(-1+V_r/mV, 0.),
          ylabel=f"membrane\npotential\n(${sympy.latex(mV)}$)")


### Synaptic variables
# Retrieves indexes of spikes in the synaptic monitor using the fact that we
# are sampling spikes and synaptic variables by the same dt
spk_index = np.in1d(synapse_mon.t, exc_mon.t[exc_mon.i == ni])
ax[2].plot(synapse_mon.t[spk_index]/ms, synapse_mon.x_S[0][spk_index], '.',
           ms=4, color='C3')
ax[2].plot(synapse_mon.t[spk_index]/ms, synapse_mon.u_S[0][spk_index], '.',
           ms=4, color='C4')
# Super-impose reconstructed solutions
time = synapse_mon.t  # time vector
tspk = Quantity(synapse_mon.t, copy=True)  # Spike times
for ts in exc_mon.t[exc_mon.i == ni]:
    tspk[time >= ts] = ts
ax[2].plot(synapse_mon.t/ms, 1 + (synapse_mon.x_S[0]-1)*exp(-(time-tspk)*Omega_d),
           '-', color='C3')
ax[2].plot(synapse_mon.t/ms, synapse_mon.u_S[0]*exp(-(time-tspk)*Omega_f),
           '-', color='C4')
# Adjust axis
pu.adjust_spines(ax[2], ['left'])
ax[2].set(xlim=(0., duration/ms), ylim=(-0.05, 1.05),
          ylabel='synaptic\nvariables\n$u_S,\,x_S$')

nspikes = np.sum(spk_index)
x_S_spike = synapse_mon.x_S[0][spk_index]
u_S_spike = synapse_mon.u_S[0][spk_index]
ax[3].vlines(synapse_mon.t[spk_index]/ms, np.zeros(nspikes),
             x_S_spike*u_S_spike/(1-u_S_spike))
pu.adjust_spines(ax[3], ['left', 'bottom'])
ax[3].set(xlim=(0., duration/ms), ylim=(-0.01, 0.62),
          yticks=np.arange(0, 0.62, 0.2), xlabel='time (ms)', ylabel='$r_S$')

pu.adjust_ylabels(ax, x_offset=-0.20)


plt.show()
```
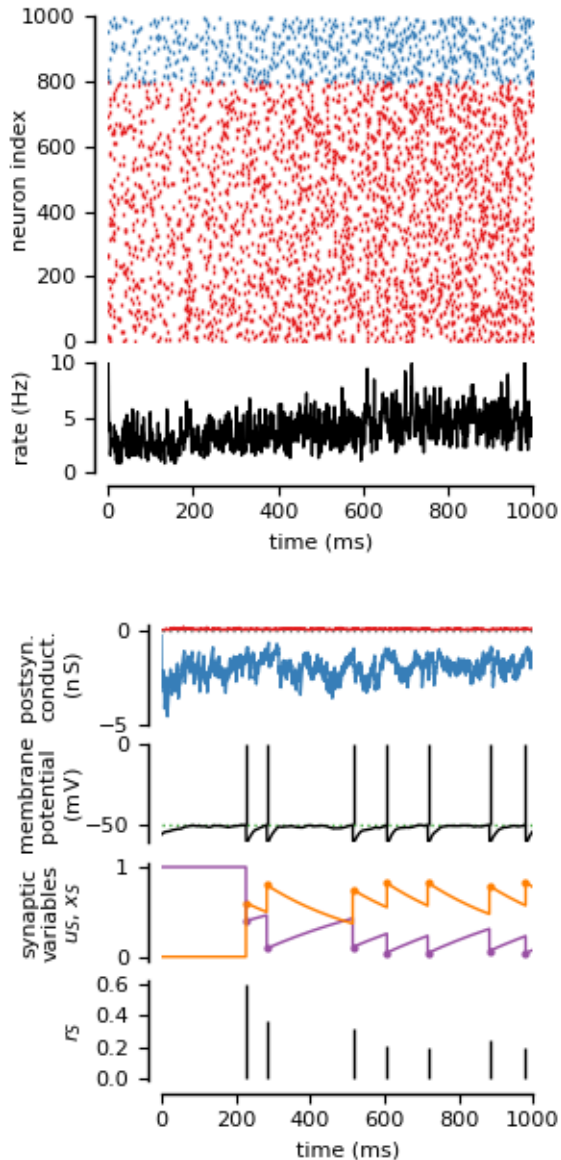
5.13.2 Example: example_2_gchi_astrocyte

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 2: Modeling of synaptically-activated astrocytes

Two astrocytes (one stochastic and the other deterministic) activated by synapses (connecting "dummy" groups of neurons) (see De Pitta' et al., 2009)

```
from brian2 import *

import plot_utils as pu
```

```python
set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"
seed(790824)  # to get identical figures for repeated runs


###############################################################################
# Model parameters
###############################################################################
### General parameters
duration = 30*second        # Total simulation time
sim_dt = 1*ms               # Integrator/sampling step


### Neuron parameters
f_0 = 0.5*Hz                # Spike rate of the "source" neurons


### Synapse parameters
rho_c = 0.001               # Synaptic vesicle-to-extracellular space volume ratio
Y_T = 500*mmolar            # Total vesicular neurotransmitter concentration
Omega_c = 40/second         # Neurotransmitter clearance rate


### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second     # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.1 * umolar          # Ca2+ affinity of SERCAs
C_T = 2*umolar              # Total cell free Ca^2+ content
rho_A = 0.18                # ER-to-cytoplasm volume ratio
Omega_C = 6/second          # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second        # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar           # IP_3 binding affinity
d_2 = 1.05*umolar           # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second     # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar         # IP_3 dissociation constant
d_5 = 0.08*umolar           # Ca^2+ activation dissociation constant
# --- Agonist-dependent IP_3 production
O_beta = 5*umolar/second    # Maximal rate of IP_3 production by PLCbeta
O_N = 0.3/umolar/second     # Agonist binding rate
Omega_N = 0.5/second        # Maximal inactivation rate
K_KC = 0.5*umolar           # Ca^2+ affinity of PKC
zeta = 10                   # Maximal reduction of receptor affinity by PKC
# --- IP_3 production
O_delta = 0.2 *umolar/second # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5 * umolar  # Inhibition constant of PLC_delta by IP_3
K_delta = 0.3*umolar        # Ca^2+ affinity of PLCdelta
# --- IP_3 degradation
Omega_5P = 0.1/second       # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.5*umolar            # Ca^2+ affinity of IP3-3K
K_3K = 1*umolar             # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second    # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 external production
F_ex = 0.09*umolar/second   # Maximal exogenous IP3 flow
I_Theta = 0.3*umolar        # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar       # Scaling factor of diffusion


###############################################################################
# Model definition
###############################################################################
defaultclock.dt = sim_dt  # Set the integration time
```

```
### "Neurons"
# (We are only interested in the activity of the synapse, so we replace the
# neurons by trivial "dummy" groups
# # Regular spiking neuron
source_neurons = NeuronGroup(1, 'dx/dt = f_0 : 1', threshold='x>1',
                             reset='x=0', method='euler')
## Dummy neuron
target_neurons = NeuronGroup(1, '')


### Synapses
# Our synapse model is trivial, we are only interested in its neurotransmitter
# release
synapses_eqs = 'dY_S/dt = -Omega_c * Y_S : mmolar (clock-driven)'
synapses_action = 'Y_S += rho_c * Y_T'
synapses = Synapses(source_neurons, target_neurons,
                    model=synapses_eqs, on_pre=synapses_action,
                    method='exact')
synapses.connect()


### Astrocytes
# We are modelling two astrocytes, the first is deterministic while the second
# displays stochastic dynamics
astro_eqs = '''
# Fraction of activated astrocyte receptors:
dGamma_A/dt = O_N * Y_S * (1 - Gamma_A) -
              Omega_N*(1 + zeta * C/(C + K_KC)) * Gamma_A : 1

# IP_3 dynamics:
dI/dt = J_beta + J_delta - J_3K - J_5P + J_ex     : mmolar
J_beta = O_beta * Gamma_A                          : mmolar/second
J_delta = O_delta/(1 + I/kappa_delta) *
                         C**2/(C**2 + K_delta**2) : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K) : mmolar/second
J_5P = Omega_5P*I                                 : mmolar/second
delta_I_bias = I - I_bias : mmolar
J_ex = -F_ex/2*(1 + tanh((abs(delta_I_bias) - I_Theta)/omega_I)) *
               sign(delta_I_bias)                 : mmolar/second
I_bias                                            : mmolar (constant)

# Ca^2+-induced Ca^2+ release:
dC/dt = J_r + J_l - J_p                  : mmolar
# IP3R de-inactivation probability
dh/dt = (h_inf - h_clipped)/tau_h *
        (1 + noise*xi*tau_h**0.5)        : 1
h_clipped = clip(h,0,1)                  : 1
J_r = (Omega_C * m_inf**3 * h_clipped**3) *
      (C_T - (1 + rho_A)*C)              : mmolar/second
J_l = Omega_L * (C_T - (1 + rho_A)*C)    : mmolar/second
J_p = O_P * C**2/(C**2 + K_P**2)         : mmolar/second
m_inf = I/(I + d_1) * C/(C + d_5)        : 1
h_inf = Q_2/(Q_2 + C)                    : 1
tau_h = 1/(O_2 * (Q_2 + C))              : second
Q_2 = d_2 * (I + d_1)/(I + d_3)          : mmolar

# Neurotransmitter concentration in the extracellular space
```

```
Y_S      : mmolar
# Noise flag
noise    : 1 (constant)
'''
# Milstein integration method for the multiplicative noise
astrocytes = NeuronGroup(2, astro_eqs, method='milstein')
astrocytes.h = 0.9  # IP3Rs are initially mostly available for CICR

# The first astrocyte is deterministic ("zero noise"), the second stochastic
astrocytes.noise = [0, 1]
# Connection between synapses and astrocytes (both astrocytes receive the
# same input from the synapse). Note that in this special case, where each
# astrocyte is only influenced by the neurotransmitter from a single synapse,
# the '(linked)' variable mechanism could be used instead. The mechanism used
# below is more general and can add the contribution of several synapses.
ecs_syn_to_astro = Synapses(synapses, astrocytes,
                            'Y_S_post = Y_S_pre : mmolar (summed)')
ecs_syn_to_astro.connect()
################################################################################
# Monitors
################################################################################
astro_mon = StateMonitor(astrocytes, variables=['Gamma_A', 'C', 'h', 'I'],
                         record=True)

################################################################################
# Simulation run
################################################################################
run(duration, report='text')

################################################################################
# Analysis and plotting
################################################################################
from matplotlib.ticker import FormatStrFormatter
plt.style.use('figures.mplstyle')

# Plot Gamma_A
fig, ax = plt.subplots(4, 1, figsize=(6.26894, 6.26894*0.66))
ax[0].plot(astro_mon.t/second, astro_mon.Gamma_A.T)
ax[0].set(xlim=(0., duration/second), ylim=[-0.05, 1.02], yticks=[0.0, 0.5, 1.0],
          ylabel=r'$\Gamma_{A}$')
# Adjust axis
pu.adjust_spines(ax[0], ['left'])

# Plot I
ax[1].plot(astro_mon.t/second, astro_mon.I.T/umolar)
ax[1].set(xlim=(0., duration/second), ylim=[-0.1, 5.0],
          yticks=arange(0.0, 5.1, 1., dtype=float),
          ylabel=r'$I$ ($\mu M$)')
ax[1].yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
ax[1].legend(['deterministic', 'stochastic'], loc='upper left')
pu.adjust_spines(ax[1], ['left'])

# Plot C
ax[2].plot(astro_mon.t/second, astro_mon.C.T/umolar)
ax[2].set(xlim=(0., duration/second), ylim=[-0.1, 1.3],
          ylabel=r'$C$ ($\mu M$)')
pu.adjust_spines(ax[2], ['left'])
```

```
# Plot h
ax[3].plot(astro_mon.t/second, astro_mon.h.T)
ax[3].set(xlim=(0., duration/second),
          ylim=[0.4, 1.02],
          ylabel='h', xlabel='time ($s$)')
pu.adjust_spines(ax[3], ['left', 'bottom'])

pu.adjust_ylabels(ax, x_offset=-0.1)

plt.show()
```
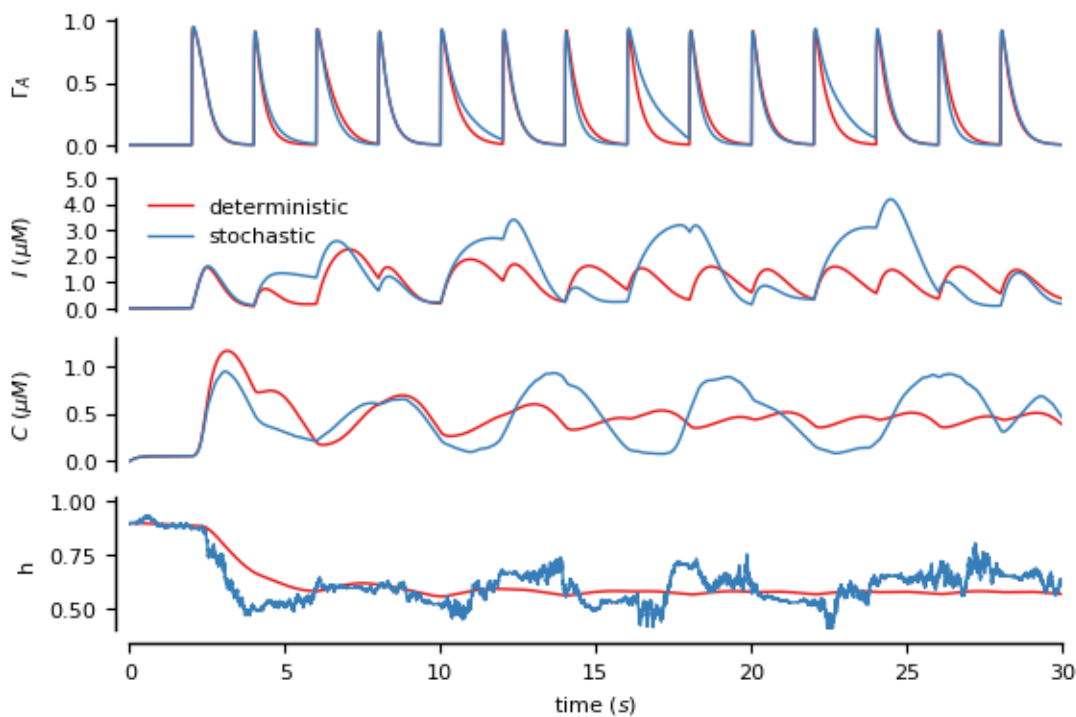

5.13.3 Example: example_3_io_synapse

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 3: Modeling of modulation of synaptic release by gliotransmission.

Three synapses: the first one without astrocyte, the remaining two respectively with open-loop and close-loop gliotransmission (see De Pitta' et al., 2011, 2016)

```
from brian2 import *

import plot_utils as pu
```

```python
set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"


################################################################################
# Model parameters
################################################################################
### General parameters
transient = 16.5*second
duration = transient + 600*ms    # Total simulation time
sim_dt = 1*ms                    # Integrator/sampling step


### Synapse parameters
rho_c = 0.005                    # Synaptic vesicle-to-extracellular space volume ratio
Y_T = 500*mmolar                 # Total vesicular neurotransmitter concentration
Omega_c = 40/second              # Neurotransmitter clearance rate
U_0__star = 0.6                  # Resting synaptic release probability
Omega_f = 3.33/second            # Synaptic facilitation rate
Omega_d = 2.0/second             # Synaptic depression rate
# --- Presynaptic receptors
O_G = 1.5/umolar/second          # Agonist binding (activating) rate
Omega_G = 0.5/(60*second)        # Agonist release (deactivating) rate


### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second          # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.05 * umolar              # Ca2+ affinity of SERCAs
C_T = 2*umolar                   # Total cell free Ca^2+ content
rho_A = 0.18                     # ER-to-cytoplasm volume ratio
Omega_C = 6/second               # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second             # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar                # IP_3 binding affinity
d_2 = 1.05*umolar                # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second          # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar              # IP_3 dissociation constant
d_5 = 0.08*umolar                # Ca^2+ activation dissociation constant
# --- IP_3 production
O_delta = 0.6*umolar/second      # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5* umolar        # Inhibition constant of PLC_delta by IP_3
K_delta = 0.1*umolar             # Ca^2+ affinity of PLCdelta
# --- IP_3 degradation
Omega_5P = 0.05/second           # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.7*umolar                 # Ca^2+ affinity of IP3-3K
K_3K = 1.0*umolar                # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second         # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 diffusion
F_ex = 2.0*umolar/second         # Maximal exogenous IP3 flow
I_Theta = 0.3*umolar             # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar            # Scaling factor of diffusion
# --- Gliotransmitter release and time course
C_Theta = 0.5*umolar             # Ca^2+ threshold for exocytosis
Omega_A = 0.6/second             # Gliotransmitter recycling rate
U_A = 0.6                        # Gliotransmitter release probability
G_T = 200*mmolar                 # Total vesicular gliotransmitter concentration
rho_e = 6.5e-4                   # Astrocytic vesicle-to-extracellular volume ratio
Omega_e = 60/second              # Gliotransmitter clearance rate
alpha = 0.0                      # Gliotransmission nature
```

```
################################################################################
# Model definition
################################################################################
defaultclock.dt = sim_dt  # Set the integration time

### "Neurons"
# We are only interested in the activity of the synapse, so we replace the
# neurons by trivial "dummy" groups
spikes = [0, 50, 100, 150, 200,
          300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400]*ms
spikes += transient  # allow for some initial transient
source_neurons = SpikeGeneratorGroup(1, np.zeros(len(spikes)), spikes)
target_neurons = NeuronGroup(1, '')


### Synapses
# Note that the synapse does not actually have any effect on the post-synaptic
# target
# Also note that for easier plotting we do not use the "event-driven" flag here,
# even though the value of u_S and x_S only needs to be updated on the arrival
# of a spike
synapses_eqs = '''
# Neurotransmitter
dY_S/dt = -Omega_c * Y_S         : mmolar (clock-driven)
# Fraction of activated presynaptic receptors
dGamma_S/dt = O_G * G_A * (1 - Gamma_S) -
              Omega_G * Gamma_S : 1 (clock-driven)
# Usage of releasable neurotransmitter per single action potential:
du_S/dt = -Omega_f * u_S        : 1 (clock-driven)
# Fraction of synaptic neurotransmitter resources available:
dx_S/dt = Omega_d *(1 - x_S)    : 1 (clock-driven)
# released synaptic neurotransmitter resources:
r_S                             : 1
# gliotransmitter concentration in the extracellular space:
G_A                             : mmolar
'''
synapses_action = '''
U_0 = (1 - Gamma_S) * U_0__star + alpha * Gamma_S
u_S += U_0 * (1 - u_S)
r_S = u_S * x_S
x_S -= r_S
Y_S += rho_c * Y_T * r_S
'''
synapses = Synapses(source_neurons, target_neurons,
                    model=synapses_eqs, on_pre=synapses_action,
                    method='exact')
# We create three synapses, only the second and third ones are modulated by astrocytes
synapses.connect(True, n=3)


### Astrocytes
# The astrocyte emits gliotransmitter when its Ca^2+ concentration crosses
# a threshold
astro_eqs = '''
# IP_3 dynamics:
dI/dt = J_delta - J_3K - J_5P + J_ex                             : mmolar
J_delta = O_delta/(1 + I/kappa_delta) * C**2/(C**2 + K_delta**2) : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K)                : mmolar/second
J_5P = Omega_5P*I                                               : mmolar/second
```

```
# Exogenous stimulation
delta_I_bias = I - I_bias              : mmolar
J_ex = -F_ex/2*(1 + tanh((abs(delta_I_bias) - I_Theta)/omega_I)) *
               sign(delta_I_bias) : mmolar/second
I_bias                                 : mmolar (constant)

# Ca^2+-induced Ca^2+ release:
dC/dt = (Omega_C * m_inf**3 * h**3 + Omega_L) * (C_T - (1 + rho_A)*C) -
        O_P * C**2/(C**2 + K_P**2) : mmolar
dh/dt = (h_inf - h)/tau_h : 1  # IP3R de-inactivation probability
m_inf = I/(I + d_1) * C/(C + d_5)   : 1
h_inf = Q_2/(Q_2 + C)               : 1
tau_h = 1/(O_2 * (Q_2 + C))         : second
Q_2 = d_2 * (I + d_1)/(I + d_3)     : mmolar
# Fraction of gliotransmitter resources available:
dx_A/dt = Omega_A * (1 - x_A)       : 1
# gliotransmitter concentration in the extracellular space:
dG_A/dt = -Omega_e*G_A              : mmolar
'''
glio_release = '''
G_A += rho_e * G_T * U_A * x_A
x_A -= U_A *  x_A
'''
# The following formulation makes sure that a "spike" is only triggered at the
# first threshold crossing -- the astrocyte is considered "refractory" (i.e.,
# not allowed to trigger another event) as long as the Ca2+ concentration
# remains above threshold
# The gliotransmitter release happens when the threshold is crossed, in Brian
# terms it can therefore be considered a "reset"
astrocyte = NeuronGroup(2, astro_eqs,
                        threshold='C>C_Theta',
                        refractory='C>C_Theta',
                        reset=glio_release,
                        method='rk4')
# Different length of stimulation
astrocyte.x_A = 1.0
astrocyte.h = 0.9
astrocyte.I = 0.4*umolar
astrocyte.I_bias = np.asarray([0.8, 1.25])*umolar

# Connection between astrocytes and the second synapse. Note that in this
# special case, where the synapse is only influenced by the gliotransmitter from
# a single astrocyte, the '(linked)' variable mechanism could be used instead.
# The mechanism used below is more general and can add the contribution of
# several astrocytes
ecs_astro_to_syn = Synapses(astrocyte, synapses,
                            'G_A_post = G_A_pre : mmolar (summed)')
# Connect second and third synapse to a different astrocyte
ecs_astro_to_syn.connect(j='i+1')

###############################################################################
# Monitors
###############################################################################
# Note that we cannot use "record=True" for synapses in C++ standalone mode --
# the StateMonitor needs to know the number of elements to record from during
# its initialization, but in C++ standalone mode, no synapses have been created
# yet. We therefore explicitly state to record from the three synapses.
```

```python
syn_mon = StateMonitor(synapses, variables=['u_S', 'x_S', 'r_S', 'Y_S'],
                       record=[0, 1, 2])
ast_mon = StateMonitor(astrocyte, variables=['C', 'G_A'], record=True)


################################################################################
# Simulation run
################################################################################
run(duration, report='text')


################################################################################
# Analysis and plotting
################################################################################
from matplotlib import cycler
plt.style.use('figures.mplstyle')

fig, ax = plt.subplots(nrows=7, ncols=1, figsize=(6.26894, 6.26894 * 1.2),
                       gridspec_kw={'height_ratios': [3, 2, 1, 1, 3, 3, 3],
                                    'top': 0.98, 'bottom': 0.08,
                                    'left': 0.15, 'right': 0.95})

## Ca^2+ traces of the two astrocytes
ax[0].plot((ast_mon.t-transient)/second, ast_mon.C[0]/umolar, '-', color='C2')
ax[0].plot((ast_mon.t-transient)/second, ast_mon.C[1]/umolar, '-', color='C3')
## Add threshold for gliotransmitter release
ax[0].plot(np.asarray([-transient/second, 0.0]),
           np.asarray([C_Theta, C_Theta])/umolar, ':', color='gray')
ax[0].set(xlim=[-transient/second, 0.0], yticks=[0., 0.4, 0.8, 1.2],
          ylabel=r'$C$ ($\mu$M)')
pu.adjust_spines(ax[0], ['left'])

## Gliotransmitter concentration in the extracellular space
ax[1].plot((ast_mon.t-transient)/second, ast_mon.G_A[0]/umolar, '-', color='C2')
ax[1].plot((ast_mon.t-transient)/second, ast_mon.G_A[1]/umolar, '-', color='C3')
ax[1].set(yticks=[0., 50., 100.], xlim=[-transient/second, 0.0],
          xlabel='time (s)', ylabel=r'$G_A$ ($\mu$M)')
pu.adjust_spines(ax[1], ['left', 'bottom'])

## Turn off one axis to display x-labeling of ax[1] correctly
ax[2].axis('off')

## Synaptic stimulation
ax[3].vlines((spikes-transient)/ms, 0, 1, clip_on=False)
ax[3].set(xlim=(0, (duration-transient)/ms))
ax[3].axis('off')

## Synaptic variables
# Use a custom cycle that uses black as the first color
prop_cycle = cycler(color='k').concat(matplotlib.rcParams['axes.prop_cycle'][2:])
ax[4].set(xlim=(0, (duration-transient)/ms), ylim=[0., 1.],
          yticks=np.arange(0, 1.1, .25), ylabel='$u_S$',
          prop_cycle=prop_cycle)
ax[4].plot((syn_mon.t-transient)/ms, syn_mon.u_S.T)
pu.adjust_spines(ax[4], ['left'])

ax[5].set(xlim=(0, (duration-transient)/ms), ylim=[-0.05, 1.],
          yticks=np.arange(0, 1.1, .25), ylabel='$x_S$',
          prop_cycle=prop_cycle)
```

```
ax[5].plot((syn_mon.t-transient)/ms, syn_mon.x_S.T)
pu.adjust_spines(ax[5], ['left'])


ax[6].set(xlim=(0, (duration-transient)/ms), ylim=(-5., 1500),
          xticks=np.arange(0, (duration-transient)/ms, 100), xlabel='time (ms)',
          yticks=[0, 500, 1000, 1500], ylabel=r'$Y_S$ ($\mu$M)',
          prop_cycle=prop_cycle)
ax[6].plot((syn_mon.t-transient)/ms, syn_mon.Y_S.T/umolar)
ax[6].legend(['no gliotransmission',
              'weak gliotransmission',
              'stronger gliotransmission'], loc='upper right')
pu.adjust_spines(ax[6], ['left', 'bottom'])

pu.adjust_ylabels(ax, x_offset=-0.11)

plt.show()
```
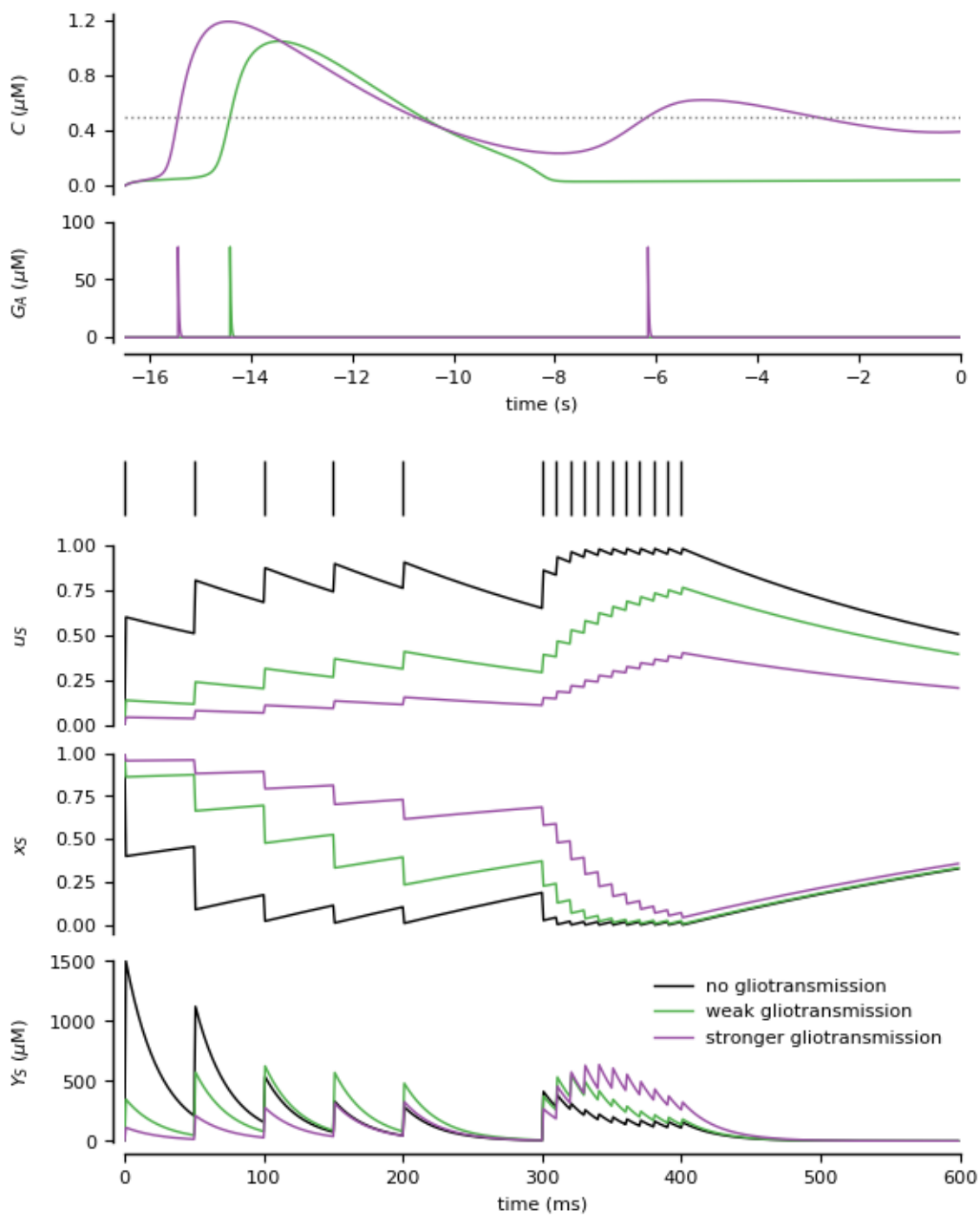
5.13.4 Example: example_4_rsmean

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 4C: Closed-loop gliotransmission.

I/O curves in terms average per-spike release vs. rate of stimulation for three synapses: one without gliotransmission, and the other two with open- and close-loop gliotransmssion.

```python
from brian2 import *

import plot_utils as pu

set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"
seed(1929)  # to get identical figures for repeated runs


################################################################################
# Model parameters
################################################################################
### General parameters
N_synapses = 100
N_astro = 2
transient = 15*second
duration = transient + 180*second  # Total simulation time
sim_dt = 1*ms                       # Integrator/sampling step


### Neuron parameters

# ### Synapse parameters
### Synapse parameters
rho_c = 0.005               # Synaptic vesicle-to-extracellular space volume ratio
Y_T = 500*mmolar            # Total vesicular neurotransmitter concentration
Omega_c = 40/second         # Neurotransmitter clearance rate
U_0__star = 0.6             # Resting synaptic release probability
Omega_f = 3.33/second       # Synaptic facilitation rate
Omega_d = 2.0/second        # Synaptic depression rate
# --- Presynaptic receptors
O_G = 1.5/umolar/second     # Agonist binding (activating) rate
Omega_G = 0.5/(60*second)   # Agonist release (deactivating) rate


### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second     # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.05 * umolar         # Ca2+ affinity of SERCAs
C_T = 2*umolar              # Total cell free Ca^2+ content
rho_A = 0.18                # ER-to-cytoplasm volume ratio
Omega_C = 6/second          # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second        # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar           # IP_3 binding affinity
d_2 = 1.05*umolar           # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second     # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar         # IP_3 dissociation constant
d_5 = 0.08*umolar           # Ca^2+ activation dissociation constant
# --- IP_3 production
# --- Agonist-dependent IP_3 production
```

(continues on next page)

```python
O_beta = 3.2*umolar/second  # Maximal rate of IP_3 production by PLCbeta
O_N = 0.3/umolar/second     # Agonist binding rate
Omega_N = 0.5/second        # Maximal inactivation rate
K_KC = 0.5*umolar           # Ca^2+ affinity of PKC
zeta = 10                   # Maximal reduction of receptor affinity by PKC
# --- Endogenous IP3 production
O_delta = 0.6*umolar/second # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5* umolar   # Inhibition constant of PLC_delta by IP_3
K_delta = 0.1*umolar        # Ca^2+ affinity of PLCdelta
# --- IP_3 degradation
Omega_5P = 0.05/second      # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.7*umolar            # Ca^2+ affinity of IP3-3K
K_3K = 1.0*umolar           # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second    # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 diffusion
F_ex = 2.0*umolar/second    # Maximal exogenous IP3 flow
I_Theta = 0.3*umolar        # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar       # Scaling factor of diffusion
# --- Gliotransmitter release and time course
C_Theta = 0.5*umolar        # Ca^2+ threshold for exocytosis
Omega_A = 0.6/second        # Gliotransmitter recycling rate
U_A = 0.6                   # Gliotransmitter release probability
G_T = 200*mmolar            # Total vesicular gliotransmitter concentration
rho_e = 6.5e-4              # Astrocytic vesicle-to-extracellular volume ratio
Omega_e = 60/second         # Gliotransmitter clearance rate
alpha = 0.0                 # Gliotransmission nature


###############################################################################
# Model definition
###############################################################################
defaultclock.dt = sim_dt  # Set the integration time

f_vals = np.logspace(-1, 2, N_synapses)*Hz
source_neurons = PoissonGroup(N_synapses, rates=f_vals)
target_neurons = NeuronGroup(N_synapses, '')


### Synapses
# Note that the synapse does not actually have any effect on the post-synaptic
# target
# Also note that for easier plotting we do not use the "event-driven" flag here,
# even though the value of u_S and x_S only needs to be updated on the arrival
# of a spike
synapses_eqs = '''
# Neurotransmitter
dY_S/dt = -Omega_c * Y_S : mmolar (clock-driven)
# Fraction of activated presynaptic receptors
dGamma_S/dt = O_G * G_A * (1 - Gamma_S) - Omega_G * Gamma_S : 1 (clock-driven)
# Usage of releasable neurotransmitter per single action potential:
du_S/dt = -Omega_f * u_S : 1 (event-driven)
# Fraction of synaptic neurotransmitter resources available for release:
dx_S/dt = Omega_d *(1 - x_S) : 1 (event-driven)
r_S : 1  # released synaptic neurotransmitter resources
G_A : mmolar  # gliotransmitter concentration in the extracellular space
'''
synapses_action = '''
U_0 = (1 - Gamma_S) * U_0__star + alpha * Gamma_S
u_S += U_0 * (1 - u_S)
```

```
r_S = u_S * x_S
x_S -= r_S
Y_S += rho_c * Y_T * r_S
'''
synapses = Synapses(source_neurons, target_neurons,
                    model=synapses_eqs, on_pre=synapses_action,
                    method='exact')
# We create three synapses per connection: only the first two are modulated by
# the astrocyte however. Note that we could also create three synapses per
# connection with a single connect call by using connect(j='i', n=3), but this
# would create synapses arranged differently (synapses connection pairs
# (0, 0), (0, 0), (0, 0), (1, 1), (1, 1), (1, 1), ..., instead of
# connections (0, 0), (1, 1), ..., (0, 0), (1, 1), ..., (0, 0), (1, 1), ...)
# making the later connection descriptions more complicated.
synapses.connect(j='i')  # closed-loop modulation
synapses.connect(j='i')  # open modulation
synapses.connect(j='i')  # no modulation
synapses.x_S = 1.0


### Astrocytes
# The astrocyte emits gliotransmitter when its Ca^2+ concentration crosses
# a threshold
astro_eqs = '''
# Fraction of activated astrocyte receptors:
dGamma_A/dt = O_N * Y_S * (1 - Gamma_A) -
            Omega_N*(1 + zeta * C/(C + K_KC)) * Gamma_A : 1

# IP_3 dynamics:
dI/dt = J_beta + J_delta - J_3K - J_5P + J_ex        : mmolar
J_beta = O_beta * Gamma_A                            : mmolar/second
J_delta = O_delta/(1 + I/kappa_delta) *
                        C**2/(C**2 + K_delta**2)     : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K)    : mmolar/second
J_5P = Omega_5P*I                                    : mmolar/second
delta_I_bias = I - I_bias : mmolar
J_ex = -F_ex/2*(1 + tanh((abs(delta_I_bias) - I_Theta)/omega_I)) *
            sign(delta_I_bias)                       : mmolar/second
I_bias                                               : mmolar (constant)

# Ca^2+-induced Ca^2+ release:
dC/dt = (Omega_C * m_inf**3 * h**3 + Omega_L) * (C_T - (1 + rho_A)*C) -
        O_P * C**2/(C**2 + K_P**2) : mmolar
dh/dt = (h_inf - h)/tau_h        : 1  # IP3R de-inactivation probability
m_inf = I/(I + d_1) * C/(C + d_5)  : 1
h_inf = Q_2/(Q_2 + C)             : 1
tau_h = 1/(O_2 * (Q_2 + C))       : second
Q_2 = d_2 * (I + d_1)/(I + d_3)   : mmolar

# Fraction of gliotransmitter resources available for release
dx_A/dt = Omega_A * (1 - x_A) : 1
# gliotransmitter concentration in the extracellular space
dG_A/dt = -Omega_e*G_A       : mmolar
# Neurotransmitter concentration in the extracellular space
Y_S                          : mmolar
'''
glio_release = '''
G_A += rho_e * G_T * U_A * x_A
```

```python
x_A -= U_A *  x_A
'''
astrocyte = NeuronGroup(N_astro*N_synapses, astro_eqs,
                            # The following formulation makes sure that a "spike" is
                            # only triggered at the first threshold crossing
                            threshold='C>C_Theta',
                            refractory='C>C_Theta',
                            # The gliotransmitter release happens when the threshold
                            # is crossed, in Brian terms it can therefore be
                            # considered a "reset"
                            reset=glio_release,
                            method='rk4')
astrocyte.h = 0.9
astrocyte.x_A = 1.0
# Only the second group of N_synapses astrocytes are activated by external stimulation
astrocyte.I_bias = (np.r_[np.zeros(N_synapses), np.ones(N_synapses)])*1.0*umolar


## Connections
ecs_syn_to_astro = Synapses(synapses, astrocyte,
                                'Y_S_post = Y_S_pre : mmolar (summed)')
# Connect the first N_synapses synapses--astrocyte pairs
ecs_syn_to_astro.connect(j='i if i < N_synapses')

ecs_astro_to_syn = Synapses(astrocyte, synapses,
                                'G_A_post = G_A_pre : mmolar (summed)')
# Connect the first N_synapses astrocytes--pairs
# (closed-loop configuration)
ecs_astro_to_syn.connect(j='i if i < N_synapses')
# Connect the second N_synapses astrocyte--synapses pairs
# (open-loop configuration)
ecs_astro_to_syn.connect(j='i if i >= N_synapses and i < 2*N_synapses')


################################################################################
# Monitors
################################################################################
syn_mon = StateMonitor(synapses, 'r_S',
                        record=np.arange(N_synapses*(N_astro+1)))


################################################################################
# Simulation run
################################################################################
run(duration, report='text')


################################################################################
# Analysis and plotting
################################################################################
plt.style.use('figures.mplstyle')

fig, ax = plt.subplots(nrows=4, ncols=1, figsize=(3.07, 3.07*1.33), sharex=False,
                        gridspec_kw={'height_ratios': [1, 3, 3, 3],
                                        'top': 0.98, 'bottom': 0.12,
                                        'left': 0.22, 'right': 0.93})

## Turn off one axis to display accordingly to the other figure in example_4_synrel.py
ax[0].axis('off')

ax[1].errorbar(f_vals/Hz, np.mean(syn_mon.r_S[2*N_synapses:], axis=1),
```

```
                  np.std(syn_mon.r_S[2*N_synapses:], axis=1),
                  fmt='o', color='black', lw=0.5)
ax[1].set(xlim=(0.08, 100), xscale='log',
          ylim=(0., 0.7),
          ylabel=r'$\langle r_S \rangle$')
pu.adjust_spines(ax[1], ['left'])

ax[2].errorbar(f_vals/Hz, np.mean(syn_mon.r_S[N_synapses:2*N_synapses], axis=1),
                  np.std(syn_mon.r_S[N_synapses:2*N_synapses], axis=1),
                  fmt='o', color='C2', lw=0.5)
ax[2].set(xlim=(0.08, 100), xscale='log',
          ylim=(0., 0.2), ylabel=r'$\langle r_S \rangle$')
pu.adjust_spines(ax[2], ['left'])

ax[3].errorbar(f_vals/Hz, np.mean(syn_mon.r_S[:N_synapses], axis=1),
                  np.std(syn_mon.r_S[:N_synapses], axis=1),
                  fmt='o', color='C3', lw=0.5)
ax[3].set(xlim=(0.08, 100), xticks=np.logspace(-1, 2, 4), xscale='log',
          ylim=(0., 0.7), xlabel='input frequency (Hz)',
          ylabel=r'$\langle r_S \rangle$')
ax[3].xaxis.set_major_formatter(ScalarFormatter())
pu.adjust_spines(ax[3], ['left', 'bottom'])

pu.adjust_ylabels(ax, x_offset=-0.2)

plt.show()
```
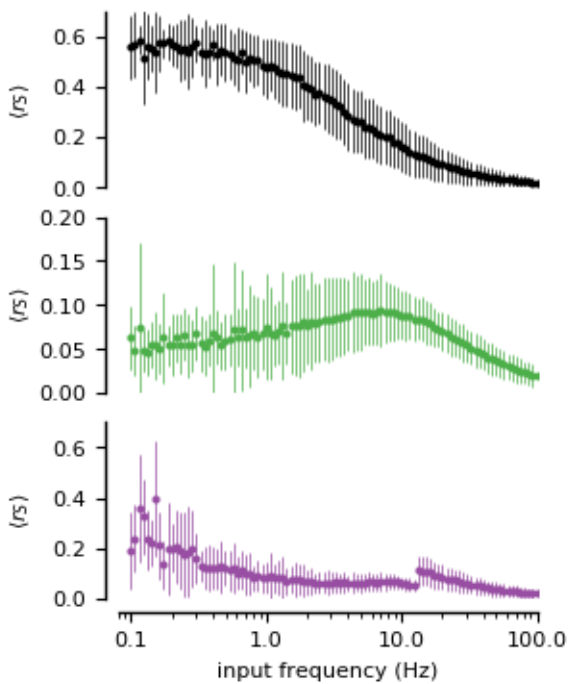
5.13.5 Example: example_4_synrel

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 4B: Closed-loop gliotransmission.

Extracellular neurotransmitter concentration (averaged across 500 synapses) for three step increases of the presynaptic rate, for three synapses: one without gliotransmission, and the other two with open- and close-loop gliotransmission.

```python
from brian2 import *

import plot_utils as pu

set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"
seed(16283)  # to get identical figures for repeated runs


################################################################################
# Model parameters
################################################################################
### General parameters
N_synapses = 500
N_astro = 2
duration = 20*second          # Total simulation time
sim_dt = 1*ms                 # Integrator/sampling step


### Neuron parameters


# ### Synapse parameters
### Synapse parameters
rho_c = 0.005                 # Synaptic vesicle-to-extracellular space volume ratio
Y_T = 500*mmolar             # Total vesicular neurotransmitter concentration
Omega_c = 40/second          # Neurotransmitter clearance rate
U_0__star = 0.6              # Resting synaptic release probability
Omega_f = 3.33/second        # Synaptic facilitation rate
Omega_d = 2.0/second         # Synaptic depression rate
# --- Presynaptic receptors
O_G = 1.5/umolar/second      # Agonist binding (activating) rate
Omega_G = 0.5/(60*second)    # Agonist release (deactivating) rate


### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second      # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.05 * umolar          # Ca2+ affinity of SERCAs
C_T = 2*umolar               # Total cell free Ca^2+ content
rho_A = 0.18                 # ER-to-cytoplasm volume ratio
Omega_C = 6/second           # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second         # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar            # IP_3 binding affinity
d_2 = 1.05*umolar            # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second      # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar          # IP_3 dissociation constant
d_5 = 0.08*umolar            # Ca^2+ activation dissociation constant
# --- IP_3 production
# --- Agonist-dependent IP_3 production
O_beta = 3.2*umolar/second   # Maximal rate of IP_3 production by PLCbeta
```

(continues on next page)

```
O_N = 0.3/umolar/second        # Agonist binding rate
Omega_N = 0.5/second           # Maximal inactivation rate
K_KC = 0.5*umolar              # Ca^2+ affinity of PKC
zeta = 10                      # Maximal reduction of receptor affinity by PKC
# --- Endogenous IP3 production
O_delta = 0.6*umolar/second    # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5* umolar      # Inhibition constant of PLC_delta by IP_3
K_delta = 0.1*umolar           # Ca^2+ affinity of PLCdelta
# --- IP_3 diffusion
F = 2*umolar/second            # GJC IP_3 permeability
I_Theta = 0.3*umolar           # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar          # Scaling factor of diffusion
# --- IP_3 degradation
Omega_5P = 0.05/second         # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.7*umolar               # Ca^2+ affinity of IP3-3K
K_3K = 1.0*umolar              # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second       # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 diffusion
F_ex = 2.0*umolar/second       # Maximal exogenous IP3 flow
I_Theta = 0.3*umolar           # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar          # Scaling factor of diffusion
# --- Gliotransmitter release and time course
C_Theta = 0.5*umolar           # Ca^2+ threshold for exocytosis
Omega_A = 0.6/second           # Gliotransmitter recycling rate
U_A = 0.6                      # Gliotransmitter release probability
G_T = 200*mmolar               # Total vesicular gliotransmitter concentration
rho_e = 6.5e-4                 # Astrocytic vesicle-to-extracellular volume ratio
Omega_e = 60/second            # Gliotransmitter clearance rate
alpha = 0.0                    # Gliotransmission nature


################################################################################
# Model definition
################################################################################
defaultclock.dt = sim_dt  # Set the integration time

# ### "Neurons"
rate_in = TimedArray([0.011, 0.11, 1.1, 11] * Hz, dt=5*second)
source_neurons = PoissonGroup(N_synapses, rates='rate_in(t)')
target_neurons = NeuronGroup(N_synapses, '')


### Synapses
# Note that the synapse does not actually have any effect on the post-synaptic
# target
# Also note that for easier plotting we do not use the "event-driven" flag here,
# even though the value of u_S and x_S only needs to be updated on the arrival
# of a spike
synapses_eqs = '''
# Neurotransmitter
dY_S/dt = -Omega_c * Y_S : mmolar (clock-driven)
# Fraction of activated presynaptic receptors
dGamma_S/dt = O_G * G_A * (1 - Gamma_S) - Omega_G * Gamma_S : 1 (clock-driven)
# Usage of releasable neurotransmitter per single action potential:
du_S/dt = -Omega_f * u_S : 1 (event-driven)
# Fraction of synaptic neurotransmitter resources available for release:
dx_S/dt = Omega_d *(1 - x_S) : 1 (event-driven)
r_S : 1  # released synaptic neurotransmitter resources
G_A : mmolar  # gliotransmitter concentration in the extracellular space
```

```python
'''
synapses_action = '''
U_0 = (1 - Gamma_S) * U_0__star + alpha * Gamma_S
u_S += U_0 * (1 - u_S)
r_S = u_S * x_S
x_S -= r_S
Y_S += rho_c * Y_T * r_S
'''
synapses = Synapses(source_neurons, target_neurons,
                    model=synapses_eqs, on_pre=synapses_action,
                    method='exact')
# We create three synapses per connection: only the first two are modulated by
# the astrocyte however. Note that we could also create three synapses per
# connection with a single connect call by using connect(j='i', n=3), but this
# would create synapses arranged differently (synapses connection pairs
# (0, 0), (0, 0), (0, 0), (1, 1), (1, 1), (1, 1), ..., instead of
# connections (0, 0), (1, 1), ..., (0, 0), (1, 1), ..., (0, 0), (1, 1), ...)
# making the later connection descriptions more complicated.
synapses.connect(j='i')  # closed-loop modulation
synapses.connect(j='i')  # open modulation
synapses.connect(j='i')  # no modulation
synapses.x_S = 1.0


### Astrocytes
# The astrocyte emits gliotransmitter when its Ca^2+ concentration crosses
# a threshold
astro_eqs = '''
# Fraction of activated astrocyte receptors:
dGamma_A/dt = O_N * Y_S * (1 - Gamma_A) -
              Omega_N*(1 + zeta * C/(C + K_KC)) * Gamma_A : 1

# IP_3 dynamics:
dI/dt = J_beta + J_delta - J_3K - J_5P + J_ex          : mmolar
J_beta = O_beta * Gamma_A                              : mmolar/second
J_delta = O_delta/(1 + I/kappa_delta) *
                        C**2/(C**2 + K_delta**2)       : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K)       : mmolar/second
J_5P = Omega_5P*I                                      : mmolar/second
delta_I_bias = I - I_bias : mmolar
J_ex = -F_ex/2*(1 + tanh((abs(delta_I_bias) - I_Theta)/omega_I)) *
              sign(delta_I_bias)                       : mmolar/second
I_bias                                                 : mmolar (constant)

# Ca^2+-induced Ca^2+ release:
dC/dt = (Omega_C * m_inf**3 * h**3 + Omega_L) * (C_T - (1 + rho_A)*C) -
        O_P * C**2/(C**2 + K_P**2) : mmolar
dh/dt = (h_inf - h)/tau_h        : 1  # IP3R de-inactivation probability
m_inf = I/(I + d_1) * C/(C + d_5)  : 1
h_inf = Q_2/(Q_2 + C)             : 1
tau_h = 1/(O_2 * (Q_2 + C))       : second
Q_2 = d_2 * (I + d_1)/(I + d_3)   : mmolar

# Fraction of gliotransmitter resources available for release
dx_A/dt = Omega_A * (1 - x_A) : 1
# gliotransmitter concentration in the extracellular space
dG_A/dt = -Omega_e*G_A        : mmolar
# Neurotransmitter concentration in the extracellular space
```

```
Y_S                              : mmolar
'''

glio_release = '''
G_A += rho_e * G_T * U_A * x_A
x_A -= U_A *  x_A
'''
astrocyte = NeuronGroup(N_astro*N_synapses, astro_eqs,
                        # The following formulation makes sure that a "spike" is
                        # only triggered at the first threshold crossing
                        threshold='C>C_Theta',
                        refractory='C>C_Theta',
                        # The gliotransmitter release happens when the threshold
                        # is crossed, in Brian terms it can therefore be
                        # considered a "reset"
                        reset=glio_release,
                        method='rk4')
astrocyte.h = 0.9
astrocyte.x_A = 1.0
# Only the second group of N_synapses astrocytes are activated by external stimulation
astrocyte.I_bias = (np.r_[np.zeros(N_synapses), np.ones(N_synapses)])*1.0*umolar

## Connections
ecs_syn_to_astro = Synapses(synapses, astrocyte,
                            'Y_S_post = Y_S_pre : mmolar (summed)')
# Connect the first N_synapses synapses--astrocyte pairs
ecs_syn_to_astro.connect(j='i if i < N_synapses')
ecs_astro_to_syn = Synapses(astrocyte, synapses,
                            'G_A_post = G_A_pre : mmolar (summed)')
# Connect the first N_synapses astrocytes--pairs (closed-loop)
ecs_astro_to_syn.connect(j='i if i < N_synapses')
# Connect the second N_synapses astrocyte--synapses pairs (open-loop)
ecs_astro_to_syn.connect(j='i if i >= N_synapses and i < 2*N_synapses')

###############################################################################
# Monitors
###############################################################################
syn_mon = StateMonitor(synapses, 'Y_S',
                       record=np.arange(N_synapses*(N_astro+1)), dt=10*ms)

###############################################################################
# Simulation run
###############################################################################
run(duration, report='text')

###############################################################################
# Analysis and plotting
###############################################################################
plt.style.use('figures.mplstyle')

fig, ax = plt.subplots(nrows=4, ncols=1, figsize=(3.07, 3.07*1.33),
                       sharex=False,
                       gridspec_kw={'height_ratios': [1, 3, 3, 3],
                                    'top': 0.98, 'bottom': 0.12,
                                    'left': 0.24, 'right': 0.95})
ax[0].semilogy(syn_mon.t/second, rate_in(syn_mon.t), '-', color='black')
ax[0].set(xlim=(0, duration/second), ylim=(0.01, 12),
          yticks=[0.01, 0.1, 1, 10], ylabel=r'$\nu_{in}$ (Hz)')
```

```python
ax[0].yaxis.set_major_formatter(ScalarFormatter())
pu.adjust_spines(ax[0], ['left'])

ax[1].plot(syn_mon.t/second,
           np.mean(syn_mon.Y_S[2*N_synapses:]/umolar, axis=0),
           '-', color='black')
ax[1].set(xlim=(0, duration/second), ylim=(-5, 260),
          yticks=np.arange(0, 260, 50),
          ylabel=r'$\langle Y_S \rangle$ ($\mu$M)')
ax[1].legend(['no gliotransmission'], loc='upper left')
pu.adjust_spines(ax[1], ['left'])

ax[2].plot(syn_mon.t/second,
           np.mean(syn_mon.Y_S[N_synapses:2*N_synapses]/umolar, axis=0),
           '-', color='C2')
ax[2].set(xlim=(0, duration/second), ylim=(-3, 150),
          yticks=np.arange(0, 151, 25),
          ylabel=r'$\langle Y_S \rangle$ ($\mu$M)')
ax[2].legend(['open-loop gliotransmission'], loc='upper left')
pu.adjust_spines(ax[2], ['left'])

ax[3].plot(syn_mon.t/second,
           np.mean(syn_mon.Y_S[:N_synapses]/umolar, axis=0),
           '-', color='C3')
ax[3].set(xlim=(0, duration/second), ylim=(-2, 150),
          xticks=np.arange(0., duration/second+1, 5.0),
          yticks=np.arange(0, 151, 25),
          xlabel='time (s)', ylabel=r'$\langle Y_S \rangle$ ($\mu$M)')
ax[3].legend(['closed-loop gliotransmission'], loc='upper left')
pu.adjust_spines(ax[3], ['left', 'bottom'])

pu.adjust_ylabels(ax, x_offset=-0.22)

plt.show()
```
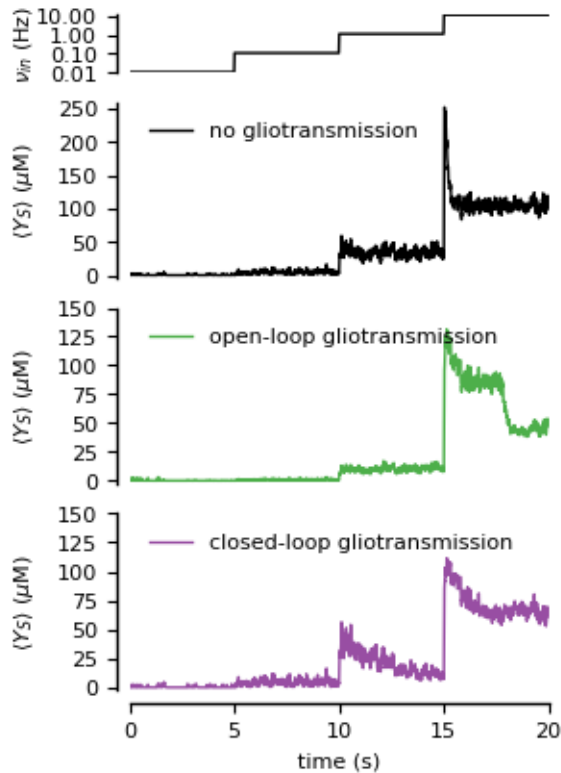
5.13.6 Example: example_5_astro_ring

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 5: Astrocytes connected in a network.

Intercellular calcium wave propagation in a ring of 50 astrocytes connected by bidirectional gap junctions (see Goldberg et al., 2010)

```python
from brian2 import *

import plot_utils as pu

set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"


################################################################################
# Model parameters
################################################################################
### General parameters
duration = 4000*second        # Total simulation time
sim_dt = 50*ms                # Integrator/sampling step

### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second       # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.05 * umolar           # Ca2+ affinity of SERCAs
C_T = 2*umolar                # Total cell free Ca^2+ content
```

(continues on next page)

```python
rho_A = 0.18                       # ER-to-cytoplasm volume ratio
Omega_C = 6/second                 # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second               # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar                  # IP_3 binding affinity
d_2 = 1.05*umolar                  # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second            # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar                # IP_3 dissociation constant
d_5 = 0.08*umolar                  # Ca^2+ activation dissociation constant
# --- IP_3 production
O_delta = 0.6*umolar/second        # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5* umolar          # Inhibition constant of PLC_delta by IP_3
K_delta = 0.1*umolar               # Ca^2+ affinity of PLCdelta
# --- IP_3 degradation
Omega_5P = 0.05/second             # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.7*umolar                   # Ca^2+ affinity of IP3-3K
K_3K = 1.0*umolar                  # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second           # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 diffusion
F_ex = 0.09*umolar/second          # Maximal exogenous IP3 flow
F = 0.09*umolar/second             # GJC IP_3 permeability
I_Theta = 0.3*umolar               # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar              # Scaling factor of diffusion


###############################################################################
# Model definition
###############################################################################
defaultclock.dt = sim_dt       # Set the integration time

### Astrocytes
astro_eqs = '''
dI/dt = J_delta - J_3K - J_5P + J_ex + J_coupling : mmolar
J_delta = O_delta/(1 + I/kappa_delta) * C**2/(C**2 + K_delta**2) : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K)               : mmolar/second
J_5P = Omega_5P*I                                              : mmolar/second
# Exogenous stimulation (rectangular wave with period of 50s and duty factor 0.4)
stimulus = int((t % (50*second))<20*second)                    : 1
delta_I_bias = I - I_bias*stimulus                             : mmolar
J_ex = -F_ex/2*(1 + tanh((abs(delta_I_bias) - I_Theta)/omega_I)) *
             sign(delta_I_bias)                                : mmolar/second
# Diffusion between astrocytes
J_coupling : mmolar/second

# Ca^2+-induced Ca^2+ release:
dC/dt = J_r + J_l - J_p                              : mmolar
dh/dt = (h_inf - h)/tau_h                            : 1
J_r = (Omega_C * m_inf**3 * h**3) * (C_T - (1 + rho_A)*C) : mmolar/second
J_l = Omega_L * (C_T - (1 + rho_A)*C)                : mmolar/second
J_p = O_P * C**2/(C**2 + K_P**2)                     : mmolar/second
m_inf = I/(I + d_1) * C/(C + d_5)                    : 1
h_inf = Q_2/(Q_2 + C)                                : 1
tau_h = 1/(O_2 * (Q_2 + C))                          : second
Q_2 = d_2 * (I + d_1)/(I + d_3)                      : mmolar

# External IP_3 drive
I_bias : mmolar (constant)
'''
```

```python
N_astro = 50 # Total number of astrocytes in the network
astrocytes = NeuronGroup(N_astro, astro_eqs, method='rk4')
# Asymmetric stimulation on the 50th cell to get some nice chaotic patterns
astrocytes.I_bias[N_astro//2] = 1.0*umolar
astrocytes.h = 0.9
# Diffusion between astrocytes
astro_to_astro_eqs = '''
delta_I = I_post - I_pre            : mmolar
J_coupling_post = -F/2 * (1 + tanh((abs(delta_I) - I_Theta)/omega_I)) *
                  sign(delta_I) : mmolar/second (summed)
'''
astro_to_astro = Synapses(astrocytes, astrocytes,
                          model=astro_to_astro_eqs)
# Couple neighboring astrocytes: two connections per astrocyte pair, as
# the above formulation will only update the I_coupling term of one of the
# astrocytes
astro_to_astro.connect('j == (i + 1) % N_pre or '
                       'j == (i + N_pre - 1) % N_pre')


################################################################################
# Monitors
################################################################################
astro_mon = StateMonitor(astrocytes, variables=['C'], record=True)


################################################################################
# Simulation run
################################################################################
run(duration, report='text')


################################################################################
# Analysis and plotting
################################################################################
plt.style.use('figures.mplstyle')

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6.26894, 6.26894 * 0.66),
                       gridspec_kw={'left': 0.1, 'bottom': 0.12})
scaling = 1.2
step = 10
ax.plot(astro_mon.t/second,
        (astro_mon.C[0:N_astro//2-1].T/astro_mon.C.max() +
         np.arange(N_astro//2-1)*scaling), color='black')
ax.plot(astro_mon.t/second, (astro_mon.C[N_astro//2:].T/astro_mon.C.max() +
                             np.arange(N_astro//2, N_astro)*scaling),
        color='black')
ax.plot(astro_mon.t/second, (astro_mon.C[N_astro//2-1].T/astro_mon.C.max() +
                             np.arange(N_astro//2-1, N_astro//2)*scaling),
        color='C0')
ax.set(xlim=(0., duration/second), ylim=(0, (N_astro+1.5)*scaling),
       xticks=np.arange(0., duration/second, 500), xlabel='time (s)',
       yticks=np.arange(0.5*scaling, (N_astro + 1.5)*scaling, step*scaling),
       yticklabels=[str(yt) for yt in np.arange(0, N_astro + 1, step)],
       ylabel='$C/C_{max}$ (cell index)')
pu.adjust_spines(ax, ['left', 'bottom'])

pu.adjust_ylabels([ax], x_offset=-0.08)
```

```
plt.show()
```
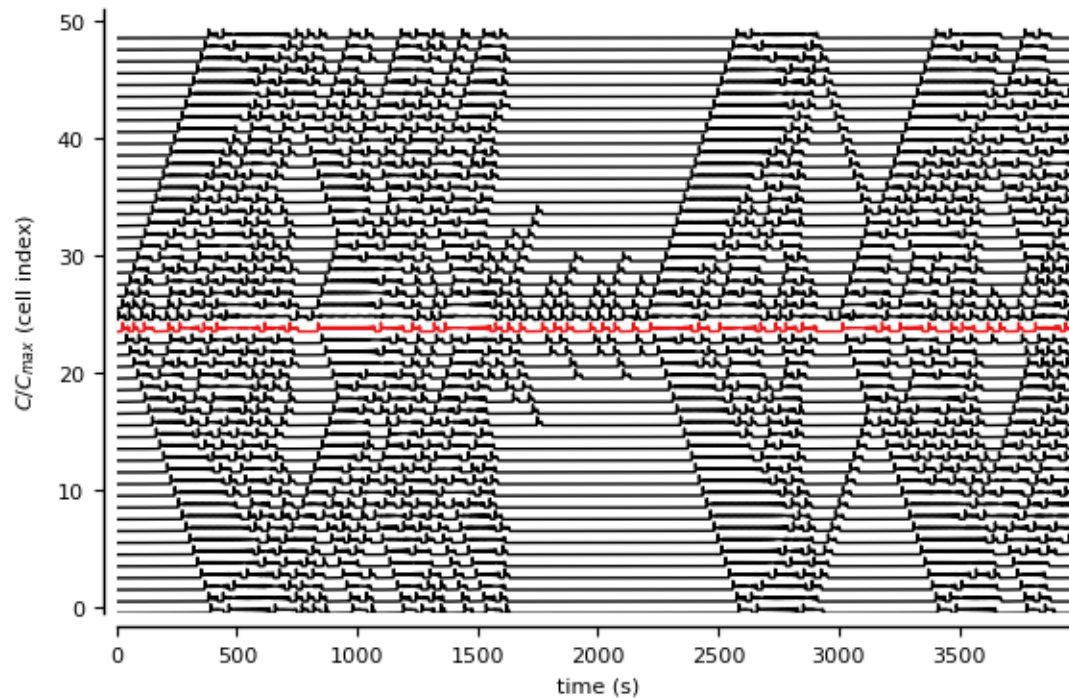

5.13.7 Example: example_6_COBA_with_astro

Modeling neuron-glia interactions with the Brian 2 simulator Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà bioRxiv 198366; doi: https://doi.org/10.1101/198366

Figure 6: Recurrent neuron-glial network.

Randomly connected COBA network (see Brunel, 2000) with excitatory synapses modulated by release-increasing glio-transmission from a randomly connected network of astrocytes.

```python
from brian2 import *

import plot_utils as pu

set_device('cpp_standalone', directory=None)  # Use fast "C++ standalone mode"
seed(28371)  # to get identical figures for repeated runs


################################################################################
# Model parameters
################################################################################
### General parameters
N_e = 3200                    # Number of excitatory neurons
N_i = 800                     # Number of inhibitory neurons
N_a = 3200                    # Number of astrocytes
```

```
## Some metrics parameters needed to establish proper connections
size = 3.75*mmeter          # Length and width of the square lattice
distance = 50*umeter        # Distance between neurons


### Neuron parameters
E_l = -60*mV                # Leak reversal potential
g_l = 9.99*nS               # Leak conductance
E_e = 0*mV                  # Excitatory synaptic reversal potential
E_i = -80*mV                # Inhibitory synaptic reversal potential
C_m = 198*pF                # Membrane capacitance
tau_e = 5*ms                # Excitatory synaptic time constant
tau_i = 10*ms               # Inhibitory synaptic time constant
tau_r = 5*ms                # Refractory period
I_ex = 100*pA               # External current
V_th = -50*mV               # Firing threshold
V_r = E_l                   # Reset potential


### Synapse parameters
rho_c = 0.005               # Synaptic vesicle-to-extracellular space volume ratio
Y_T = 500.*mmolar           # Total vesicular neurotransmitter concentration
Omega_c = 40/second         # Neurotransmitter clearance rate
U_0__star = 0.6             # Resting synaptic release probability
Omega_f = 3.33/second       # Synaptic facilitation rate
Omega_d = 2.0/second        # Synaptic depression rate
w_e = 0.05*nS               # Excitatory synaptic conductance
w_i = 1.0*nS                # Inhibitory synaptic conductance
# --- Presynaptic receptors
O_G = 1.5/umolar/second     # Agonist binding (activating) rate
Omega_G = 0.5/(60*second)   # Agonist release (deactivating) rate


### Astrocyte parameters
# ---  Calcium fluxes
O_P = 0.9*umolar/second     # Maximal Ca^2+ uptake rate by SERCAs
K_P = 0.05*umolar           # Ca2+ affinity of SERCAs
C_T = 2*umolar              # Total cell free Ca^2+ content
rho_A = 0.18                # ER-to-cytoplasm volume ratio
Omega_C = 6/second          # Maximal rate of Ca^2+ release by IP_3Rs
Omega_L = 0.1/second        # Maximal rate of Ca^2+ leak from the ER
# --- IP_3R kinectics
d_1 = 0.13*umolar           # IP_3 binding affinity
d_2 = 1.05*umolar           # Ca^2+ inactivation dissociation constant
O_2 = 0.2/umolar/second     # IP_3R binding rate for Ca^2+ inhibition
d_3 = 0.9434*umolar         # IP_3 dissociation constant
d_5 = 0.08*umolar           # Ca^2+ activation dissociation constant
# --- IP_3 production
# --- Agonist-dependent IP_3 production
O_beta = 0.5*umolar/second  # Maximal rate of IP_3 production by PLCbeta
O_N = 0.3/umolar/second     # Agonist binding rate
Omega_N = 0.5/second        # Maximal inactivation rate
K_KC = 0.5*umolar           # Ca^2+ affinity of PKC
zeta = 10                   # Maximal reduction of receptor affinity by PKC
# --- Endogenous IP3 production
O_delta = 1.2*umolar/second # Maximal rate of IP_3 production by PLCdelta
kappa_delta = 1.5*umolar    # Inhibition constant of PLC_delta by IP_3
K_delta = 0.1*umolar        # Ca^2+ affinity of PLCdelta
# --- IP_3 degradation
```

```
Omega_5P = 0.05/second          # Maximal rate of IP_3 degradation by IP-5P
K_D = 0.7*umolar                # Ca^2+ affinity of IP3-3K
K_3K = 1.0*umolar               # IP_3 affinity of IP_3-3K
O_3K = 4.5*umolar/second        # Maximal rate of IP_3 degradation by IP_3-3K
# --- IP_3 diffusion
F = 0.09*umolar/second          # GJC IP_3 permeability
I_Theta = 0.3*umolar            # Threshold gradient for IP_3 diffusion
omega_I = 0.05*umolar           # Scaling factor of diffusion
# --- Gliotransmitter release and time course
C_Theta = 0.5*umolar            # Ca^2+ threshold for exocytosis
Omega_A = 0.6/second            # Gliotransmitter recycling rate
U_A = 0.6                       # Gliotransmitter release probability
G_T = 200*mmolar                # Total vesicular gliotransmitter concentration
rho_e = 6.5e-4                  # Astrocytic vesicle-to-extracellular volume ratio
Omega_e = 60/second             # Gliotransmitter clearance rate
alpha = 0.0                     # Gliotransmission nature


################################################################################
# Define HF stimulus
################################################################################
stimulus = TimedArray([1.0, 1.2, 1.0, 1.0], dt=2*second)


################################################################################
# Simulation time (based on the stimulus)
################################################################################
duration = 8*second            # Total simulation time


################################################################################
# Model definition
################################################################################
### Neurons
neuron_eqs = '''
dv/dt = (g_l*(E_l-v) + g_e*(E_e-v) + g_i*(E_i-v) + I_ex*stimulus(t))/C_m : volt␣
↪(unless refractory)
dg_e/dt = -g_e/tau_e : siemens  # post-synaptic excitatory conductance
dg_i/dt = -g_i/tau_i : siemens  # post-synaptic inhibitory conductance
# Neuron position in space
x : meter (constant)
y : meter (constant)
'''
neurons = NeuronGroup(N_e + N_i, model=neuron_eqs,
                      threshold='v>V_th', reset='v=V_r',
                      refractory='tau_r', method='euler')
exc_neurons = neurons[:N_e]
inh_neurons = neurons[N_e:]
# Arrange excitatory neurons in a grid
N_rows = int(sqrt(N_e))
N_cols = N_e//N_rows
grid_dist = (size / N_cols)
exc_neurons.x = '(i // N_rows)*grid_dist - N_rows/2.0*grid_dist'
exc_neurons.y = '(i % N_rows)*grid_dist - N_cols/2.0*grid_dist'
# Random initial membrane potential values and conductances
neurons.v = 'E_l + rand()*(V_th-E_l)'
neurons.g_e = 'rand()*w_e'
neurons.g_i = 'rand()*w_i'

### Synapses
```

```python
synapses_eqs = '''
# Neurotransmitter
dY_S/dt = -Omega_c * Y_S                                   : mmolar (clock-driven)
# Fraction of activated presynaptic receptors
dGamma_S/dt = O_G * G_A * (1 - Gamma_S) - Omega_G * Gamma_S : 1 (clock-driven)
# Usage of releasable neurotransmitter per single action potential:
du_S/dt = -Omega_f * u_S                                   : 1 (event-driven)
# Fraction of synaptic neurotransmitter resources available for release:
dx_S/dt = Omega_d *(1 - x_S)                               : 1 (event-driven)
U_0                                                        : 1
# released synaptic neurotransmitter resources:
r_S                                                        : 1
# gliotransmitter concentration in the extracellular space:
G_A                                                        : mmolar
# which astrocyte covers this synapse ?
astrocyte_index : integer (constant)
'''
synapses_action = '''
U_0 = (1 - Gamma_S) * U_0__star + alpha * Gamma_S
u_S += U_0 * (1 - u_S)
r_S = u_S * x_S
x_S -= r_S
Y_S += rho_c * Y_T * r_S
'''
exc_syn = Synapses(exc_neurons, neurons, model=synapses_eqs,
                   on_pre=synapses_action+'g_e_post += w_e*r_S',
                   method='exact')
exc_syn.connect(True, p=0.05)
exc_syn.x_S = 1.0
inh_syn = Synapses(inh_neurons, neurons, model=synapses_eqs,
                   on_pre=synapses_action+'g_i_post += w_i*r_S',
                   method='exact')
inh_syn.connect(True, p=0.2)
inh_syn.x_S = 1.0
# Connect excitatory synapses to an astrocyte depending on the position of the
# post-synaptic neuron
N_rows_a = int(sqrt(N_a))
N_cols_a = N_a/N_rows_a
grid_dist = size / N_rows_a
exc_syn.astrocyte_index = ('int(x_post/grid_dist) + '
                           'N_cols_a*int(y_post/grid_dist)')

### Astrocytes
# The astrocyte emits gliotransmitter when its Ca^2+ concentration crosses
# a threshold
astro_eqs = '''
# Fraction of activated astrocyte receptors:
dGamma_A/dt = O_N * Y_S * (1 - clip(Gamma_A,0,1)) -
              Omega_N*(1 + zeta * C/(C + K_KC)) * clip(Gamma_A,0,1) : 1
# Intracellular IP_3
dI/dt = J_beta + J_delta - J_3K - J_5P + J_coupling         : mmolar
J_beta = O_beta * Gamma_A                                   : mmolar/second
J_delta = O_delta/(1 + I/kappa_delta) * C**2/(C**2 + K_delta**2) : mmolar/second
J_3K = O_3K * C**4/(C**4 + K_D**4) * I/(I + K_3K)           : mmolar/second
J_5P = Omega_5P*I                                           : mmolar/second
# Diffusion between astrocytes:
J_coupling                                                 : mmolar/second
```

```
# Ca^2+-induced Ca^2+ release:
dC/dt = J_r + J_l - J_p                                    : mmolar
dh/dt = (h_inf - h)/tau_h                                  : 1
J_r = (Omega_C * m_inf**3 * h**3) * (C_T - (1 + rho_A)*C) : mmolar/second
J_l = Omega_L * (C_T - (1 + rho_A)*C)                      : mmolar/second
J_p = O_P * C**2/(C**2 + K_P**2)                           : mmolar/second
m_inf = I/(I + d_1) * C/(C + d_5)                          : 1
h_inf = Q_2/(Q_2 + C)                                      : 1
tau_h = 1/(O_2 * (Q_2 + C))                                : second
Q_2 = d_2 * (I + d_1)/(I + d_3)                            : mmolar


# Fraction of gliotransmitter resources available for release:
dx_A/dt = Omega_A * (1 - x_A) : 1
# gliotransmitter concentration in the extracellular space:
dG_A/dt = -Omega_e*G_A        : mmolar
# Neurotransmitter concentration in the extracellular space:
Y_S                           : mmolar
# The astrocyte position in space
x : meter (constant)
y : meter (constant)
'''

glio_release = '''
G_A += rho_e * G_T * U_A * x_A
x_A -= U_A *  x_A
'''
astrocytes = NeuronGroup(N_a, astro_eqs,
                         # The following formulation makes sure that a "spike" is
                         # only triggered at the first threshold crossing
                         threshold='C>C_Theta',
                         refractory='C>C_Theta',
                         # The gliotransmitter release happens when the threshold
                         # is crossed, in Brian terms it can therefore be
                         # considered a "reset"
                         reset=glio_release,
                         method='rk4',
                         dt=1e-2*second)
# Arrange astrocytes in a grid
astrocytes.x = '(i // N_rows_a)*grid_dist - N_rows_a/2.0*grid_dist'
astrocytes.y = '(i % N_rows_a)*grid_dist - N_cols_a/2.0*grid_dist'
# Add random initialization
astrocytes.C = 0.01*umolar
astrocytes.h = 0.9
astrocytes.I = 0.01*umolar
astrocytes.x_A = 1.0


ecs_astro_to_syn = Synapses(astrocytes, exc_syn,
                            'G_A_post = G_A_pre : mmolar (summed)')
ecs_astro_to_syn.connect('i == astrocyte_index_post')
ecs_syn_to_astro = Synapses(exc_syn, astrocytes,
                            'Y_S_post = Y_S_pre/N_incoming : mmolar (summed)')
ecs_syn_to_astro.connect('astrocyte_index_pre == j')
# Diffusion between astrocytes
astro_to_astro_eqs = '''
delta_I = I_post - I_pre           : mmolar
J_coupling_post = -(1 + tanh((abs(delta_I) - I_Theta)/omega_I))*
                sign(delta_I)*F/2 : mmolar/second (summed)
'''
```

```
astro_to_astro = Synapses(astrocytes, astrocytes,
                          model=astro_to_astro_eqs)
# Connect to all astrocytes less than 75um away
# (about 4 connections per astrocyte)
astro_to_astro.connect('i != j and '
                       'sqrt((x_pre-x_post)**2 +'
                       '     (y_pre-y_post)**2) < 75*um')


################################################################################
# Monitors
################################################################################
# Note that we could use a single monitor for all neurons instead, but this
# way plotting is a bit easier in the end
exc_mon = SpikeMonitor(exc_neurons)
inh_mon = SpikeMonitor(inh_neurons)
ast_mon = SpikeMonitor(astrocytes)


################################################################################
# Simulation run
################################################################################
run(duration, report='text')


################################################################################
# Plot of Spiking activity
################################################################################
plt.style.use('figures.mplstyle')

fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True, figsize=(6.26894, 6.26894*0.8),
                       gridspec_kw={'height_ratios': [1, 6, 2],
                                    'left': 0.12, 'top': 0.97})
time_range = np.linspace(0, duration/second, int(duration/second*100))*second
ax[0].plot(time_range, I_ex*stimulus(time_range)/pA, 'k')
ax[0].set(xlim=(0, duration/second), ylim=(98, 122),
          yticks=[100, 120], ylabel='$I_{ex}$ (pA)')
pu.adjust_spines(ax[0], ['left'])

## We only plot a fraction of the spikes
fraction = 4
ax[1].plot(exc_mon.t[exc_mon.i <= N_e//fraction]/second,
           exc_mon.i[exc_mon.i <= N_e//fraction], '|', color='C0')
ax[1].plot(inh_mon.t[inh_mon.i <= N_i//fraction]/second,
           inh_mon.i[inh_mon.i <= N_i//fraction]+N_e//fraction, '|', color='C1')
ax[1].plot(ast_mon.t[ast_mon.i <= N_a//fraction]/second,
           ast_mon.i[ast_mon.i <= N_a//fraction]+(N_e+N_i)//fraction,
           '|', color='C2')
ax[1].set(xlim=(0, duration/second), ylim=[0, (N_e+N_i+N_a)//fraction],
          yticks=np.arange(0, (N_e+N_i+N_a)//fraction+1, 250),
          ylabel='cell index')
pu.adjust_spines(ax[1], ['left'])

# Generate frequencies
bin_size = 1*ms
spk_count, bin_edges = np.histogram(np.r_[exc_mon.t/second, inh_mon.t/second],
                                    int(duration/bin_size))
rate = 1.0*spk_count/(N_e + N_i)/bin_size/Hz
rate[rate<0.001] = 0.001 # Fix 0 lower bound for log scale
ax[2].semilogy(bin_edges[:-1], rate, '-', color='k')
```

```
pu.adjust_spines(ax[2], ['left', 'bottom'])
ax[2].set(xlim=(0, duration/second), ylim=(0.1, 150),
          xticks=np.arange(0, 9), yticks=[0.1, 1, 10, 100],
          xlabel='time (s)', ylabel='rate (Hz)')
ax[2].get_yaxis().set_major_formatter(ScalarFormatter())

pu.adjust_ylabels(ax, x_offset=-0.11)

plt.show()
```
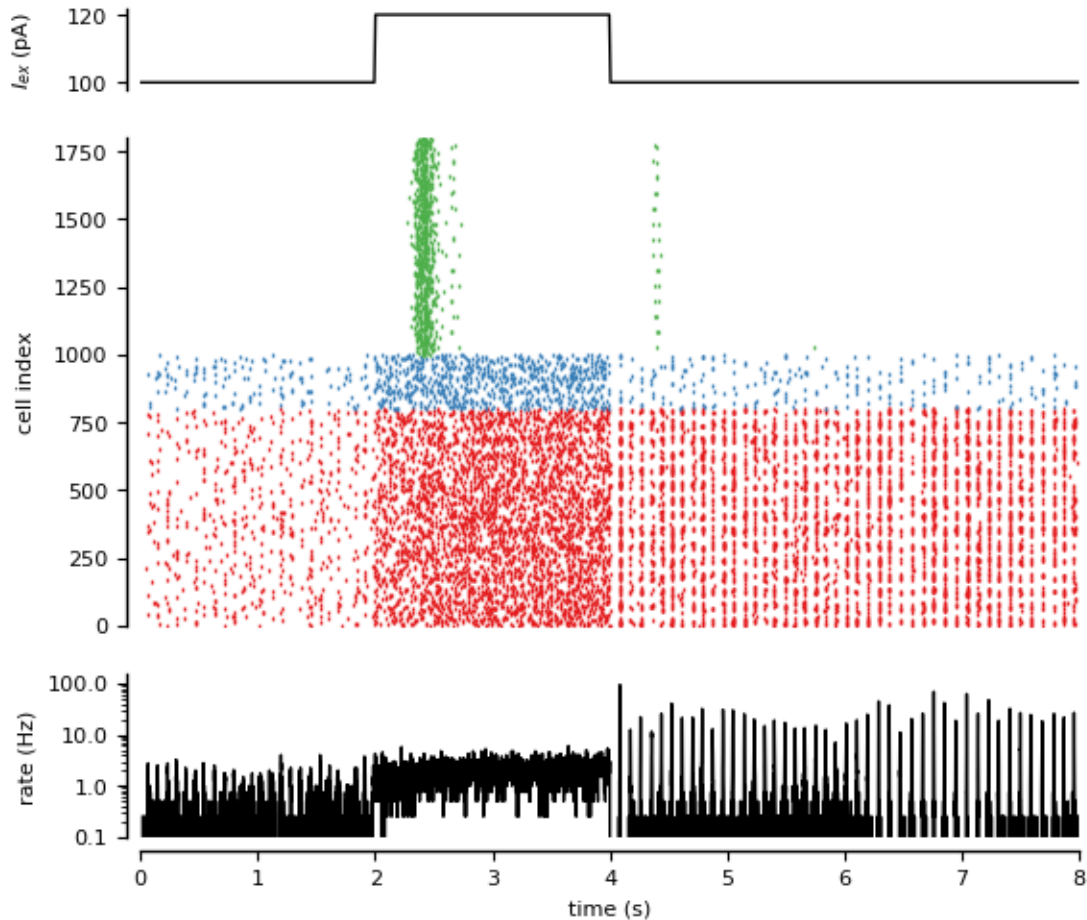

5.13.8 Example: plot_utils

Module with useful functions for making publication-ready plots.

```
def adjust_spines(ax, spines, position=5):
    """
    Set custom visibility and position of axes

    ax        : Axes
     Axes handle
```

```
    spines   : List
     String list of 'left', 'bottom', 'right', 'top' spines to show
    position : Integer
     Number of points for position of axis
    """
    for loc, spine in ax.spines.items():
        if loc in spines:
            spine.set_position(('outward', position))
        else:
            spine.set_color('none')  # don't draw spine

    # turn off ticks where there is no spine
    if 'left' in spines:
        ax.yaxis.set_ticks_position('left')
    elif 'right' in spines:
        ax.yaxis.set_ticks_position('right')
    else:
        # no yaxis ticks
        ax.yaxis.set_ticks([])
        ax.tick_params(axis='y', which='both', left='off', right='off')

    if 'bottom' in spines:
        ax.xaxis.set_ticks_position('bottom')
    elif 'top' in spines:
        ax.xaxis.set_ticks_position('top')
    else:
        # no xaxis ticks
        ax.xaxis.set_ticks([])
        ax.tick_params(axis='x', which='both', bottom='off', top='off')


def adjust_ylabels(ax,x_offset=0):
    '''
    Scan all ax list and identify the outmost y-axis position.
    Setting all the labels to that position + x_offset.
    '''

    xc = 0.0
    for a in ax:
        xc = min(xc, (a.yaxis.get_label()).get_position()[0])

    for a in ax:
        a.yaxis.set_label_coords(xc + x_offset,
                                 (a.yaxis.get_label()).get_position()[1])
```

5.13.9 README.md

```
These Brian scripts reproduce the figures from the following preprint:

Modeling neuron-glia interactions with the Brian 2 simulator
Marcel Stimberg, Dan F. M. Goodman, Romain Brette, Maurizio De Pittà
bioRxiv 198366; doi: https://doi.org/10.1101/198366

Each file can be run individually to reproduce the respective figure. Note that
```

```
most files use the [standalone mode](http://brian2.readthedocs.io/en/stable/user/
↪computation.html#standalone-code-generation)
for faster simulation. If your setup does not support this mode, you can instead
fallback to the runtime mode by removing the `set_device('cpp_standalone)` line.

Note that example 6 ("recurrent neuron-glial network") takes a relatively long
time (~15min on a reasonably fast desktop machine) to run.
```

5.13.10 figures.mplstyle

```
axes.linewidth : 1
xtick.labelsize : 8
ytick.labelsize : 8
axes.labelsize : 8
lines.linewidth : 1
lines.markersize : 2
legend.frameon : False
legend.fontsize : 8
axes.prop_cycle : cycler(color=['e41a1c', '377eb8', '4daf4a', '984ea3', 'ff7f00',
↪'ffff33'])
```

5.14 multiprocessing

5.14.1 Example: 01_using_cython

Parallel processes using Cython

This example use multiprocessing to run several simulations in parallel. The code is using the default runtime mode (and Cython compilation, if possible).

The `numb_proc` variable set the number of processes. `run_sim` is just a toy example that creates a single neuron and connects a *StateMonitor* to record the voltage.

For more details see the github issue 1154:

```python
import os
import multiprocessing

from brian2 import *


def run_sim(tau):
    pid = os.getpid()
    print(f'RUNNING {pid}')
    G = NeuronGroup(1, 'dv/dt = -v/tau : 1', method='exact')
    G.v = 1
    mon = StateMonitor(G, 'v', record=0)
    run(100*ms)
    print(f'FINISHED {pid}')
    return mon.t/ms, mon.v[0]
```

```python
if __name__ == "__main__":
    num_proc = 4

    tau_values = np.arange(10)*ms + 5*ms
    with multiprocessing.Pool(num_proc) as p:
        results = p.map(run_sim, tau_values)

    for tau_value, (t, v) in zip(tau_values, results):
        plt.plot(t, v, label=str(tau_value))
    plt.legend()
    plt.show()
```
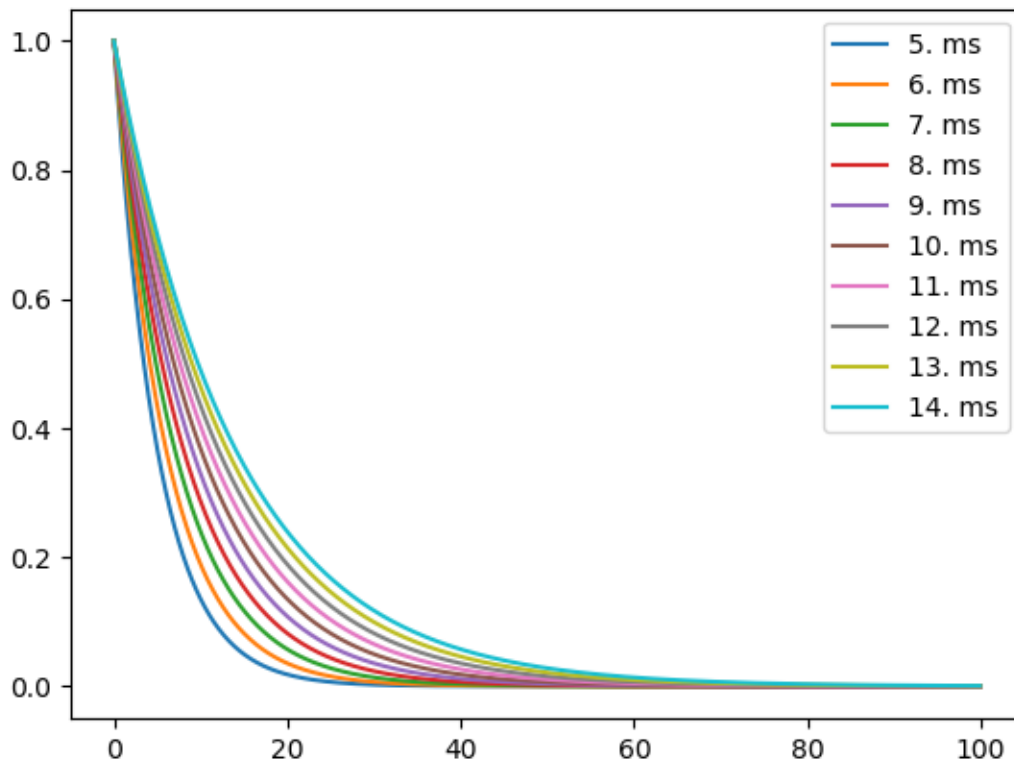
5.14.2 Example: 02_using_standalone

Parallel processes using standalone mode

This example use multiprocessing to run several simulations in parallel. The code is using the C++ standalone mode to compile and execute the code.

The generated code is stored in a `standalone{pid}` directory, with `pid` being the id of each process.

Note that the *set_device()* call should be in the `run_sim` function.

By moving the *set_device()* line into the parallelised function, it creates one C++ standalone device per process. The `device.reinit()` needs to be called` if you are running multiple simulations per process (there are 10 tau values and num_proc = 4).

Each simulation uses it's own code folder to generate the code for the simulation, controlled by the directory keyword to the set_device call. By setting `directory=None`, a temporary folder with random name is created. This way, each simulation uses a different folder for code generation and there is nothing shared between the parallel processes.

If you don't set the directory argument, it defaults to `directory="output"`. In that case each process would use the same files to try to generate and compile your simulation, which would lead to compile/execution errors.

Setting `directory=f"standalone{pid}"` is even better than using `directory=None` in this case. That is, giving each parallel process *it's own directory to work on*. This way you avoid the problem of multiple processes working on the same code directories. But you also don't need to recompile the entire project at each simulation. What happens is that in the generated code in two consecutive simulations in a single process will only differ slightly (in this case only the tau parameter). The compiler will therefore only recompile the file that has changed and not the entire project.

The `numb_proc` sets the number of processes. `run_sim` is just a toy example that creates a single neuron and connects a *StateMonitor* to record the voltage.

For more details see the discussion in the Brian forum.

```python
import os
import multiprocessing
from time import time as wall_time
from os import system
from brian2 import *

def run_sim(tau):
    pid = os.getpid()
    directory = f"standalone{pid}"
    set_device('cpp_standalone', directory=directory)
    print(f'RUNNING {pid}')

    G = NeuronGroup(1, 'dv/dt = -v/tau : 1', method='euler')
    G.v = 1

    mon = StateMonitor(G, 'v', record=0)
    net = Network()
    net.add(G, mon)
    net.run(100 * ms)
    res = (mon.t/ms, mon.v[0])

    device.reinit()

    print(f'FINISHED {pid}')
    return res
```

```python
if __name__ == "__main__":
    start_time = wall_time()

    num_proc = 4
    tau_values = np.arange(10)*ms + 5*ms
    with multiprocessing.Pool(num_proc) as p:
        results = p.map(run_sim, tau_values)

    print(f"Done in {wall_time() - start_time:10.3f}")

    for tau_value, (t, v) in zip(tau_values, results):
        plt.plot(t, v, label=str(tau_value))
    plt.legend()
    plt.show()
```
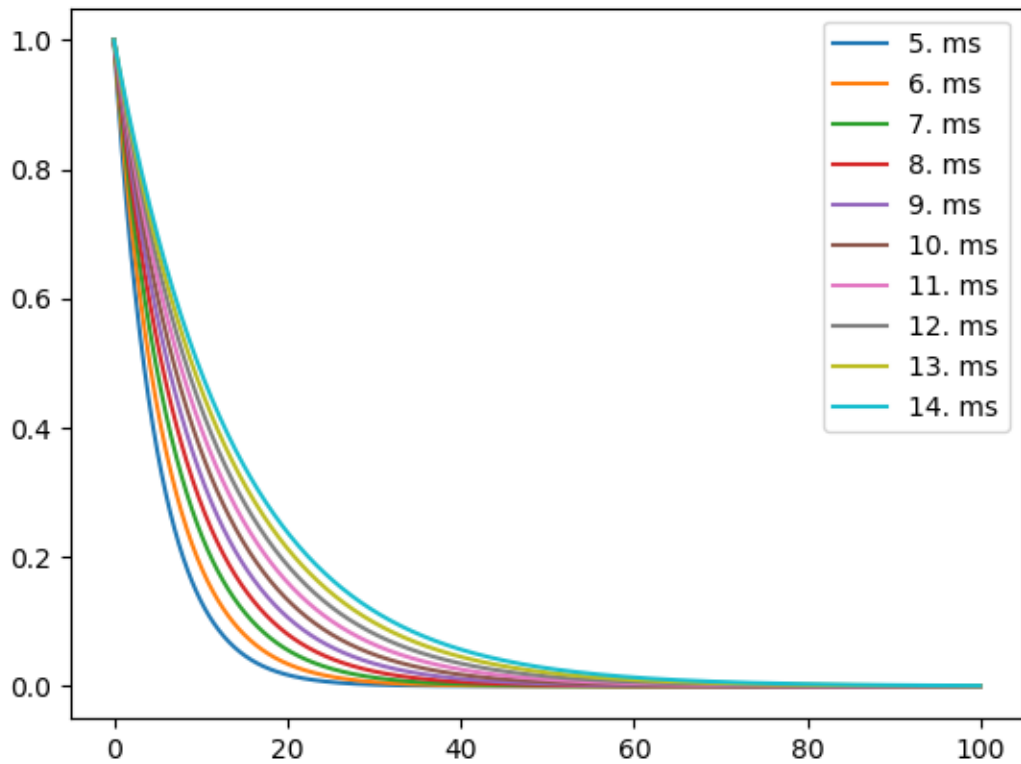
5.14.3 Example: 03_standalone_joblib

This example use C++ standalone mode for the simulation and the joblib library to parallelize the code. See the previous example (02_using_standalone.py) for more explanations.

```python
from joblib import Parallel, delayed
from time import time as wall_time
from brian2 import *
import os


def run_sim(tau):
    pid = os.getpid()
    directory = f"standalone{pid}"
    set_device('cpp_standalone', directory=directory)
    print(f'RUNNING {pid}')

    G = NeuronGroup(1, 'dv/dt = -v/tau : 1', method='euler')
    G.v = 1

    mon = StateMonitor(G, 'v', record=0)
    net = Network()
    net.add(G, mon)
    net.run(100 * ms)
    res = (mon.t/ms, mon.v[0])

    device.reinit()

    print(f'FINISHED {pid}')
    return res


if __name__ == "__main__":
    start_time = wall_time()

    n_jobs = 4
    tau_values = np.arange(10)*ms + 5*ms

    results = Parallel(n_jobs=n_jobs)(map(delayed(run_sim), tau_values))

    print(f"Done in {wall_time() - start_time:10.3f}")

    for tau_value, (t, v) in zip(tau_values, results):
        plt.plot(t, v, label=str(tau_value))
    plt.legend()
    plt.show()
```
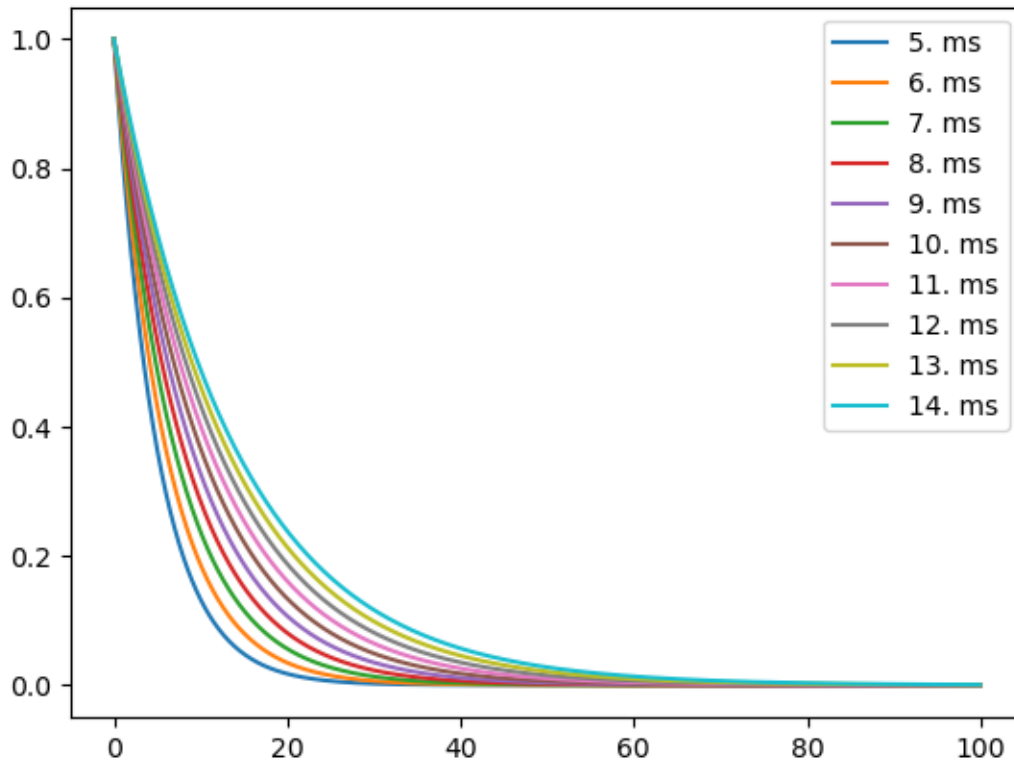
5.15 standalone

5.15.1 Example: STDP_standalone

Spike-timing dependent plasticity. Adapted from Song, Miller and Abbott (2000) and Song and Abbott (2001).

This example is modified from `synapses_STDP.py` and writes a standalone C++ project in the directory `STDP_standalone`.

```python
from brian2 import *

set_device('cpp_standalone', directory='STDP_standalone')

N = 1000
taum = 10*ms
taupre = 20*ms
taupost = taupre
Ee = 0*mV
vt = -54*mV
vr = -60*mV
El = -74*mV
```

```python
taue = 5*ms
F = 15*Hz
gmax = .01
dApre = .01
dApost = -dApre * taupre / taupost * 1.05
dApost *= gmax
dApre *= gmax

eqs_neurons = '''
dv/dt = (ge * (Ee-v) + El - v) / taum : volt
dge/dt = -ge / taue : 1
'''

input = PoissonGroup(N, rates=F)
neurons = NeuronGroup(1, eqs_neurons, threshold='v>vt', reset='v = vr',
                      method='euler')
S = Synapses(input, neurons,
             '''w : 1
                dApre/dt = -Apre / taupre : 1 (event-driven)
                dApost/dt = -Apost / taupost : 1 (event-driven)''',
             on_pre='''ge += w
                    Apre += dApre
                    w = clip(w + Apost, 0, gmax)''',
             on_post='''Apost += dApost
                     w = clip(w + Apre, 0, gmax)''',
             )
S.connect()
S.w = 'rand() * gmax'
mon = StateMonitor(S, 'w', record=[0, 1])
s_mon = SpikeMonitor(input)

run(100*second, report='text')

subplot(311)
plot(S.w / gmax, '.k')
ylabel('Weight / gmax')
xlabel('Synapse index')
subplot(312)
hist(S.w / gmax, 20)
xlabel('Weight / gmax')
subplot(313)
plot(mon.t/second, mon.w.T/gmax)
xlabel('Time (s)')
ylabel('Weight / gmax')
tight_layout()
show()
```
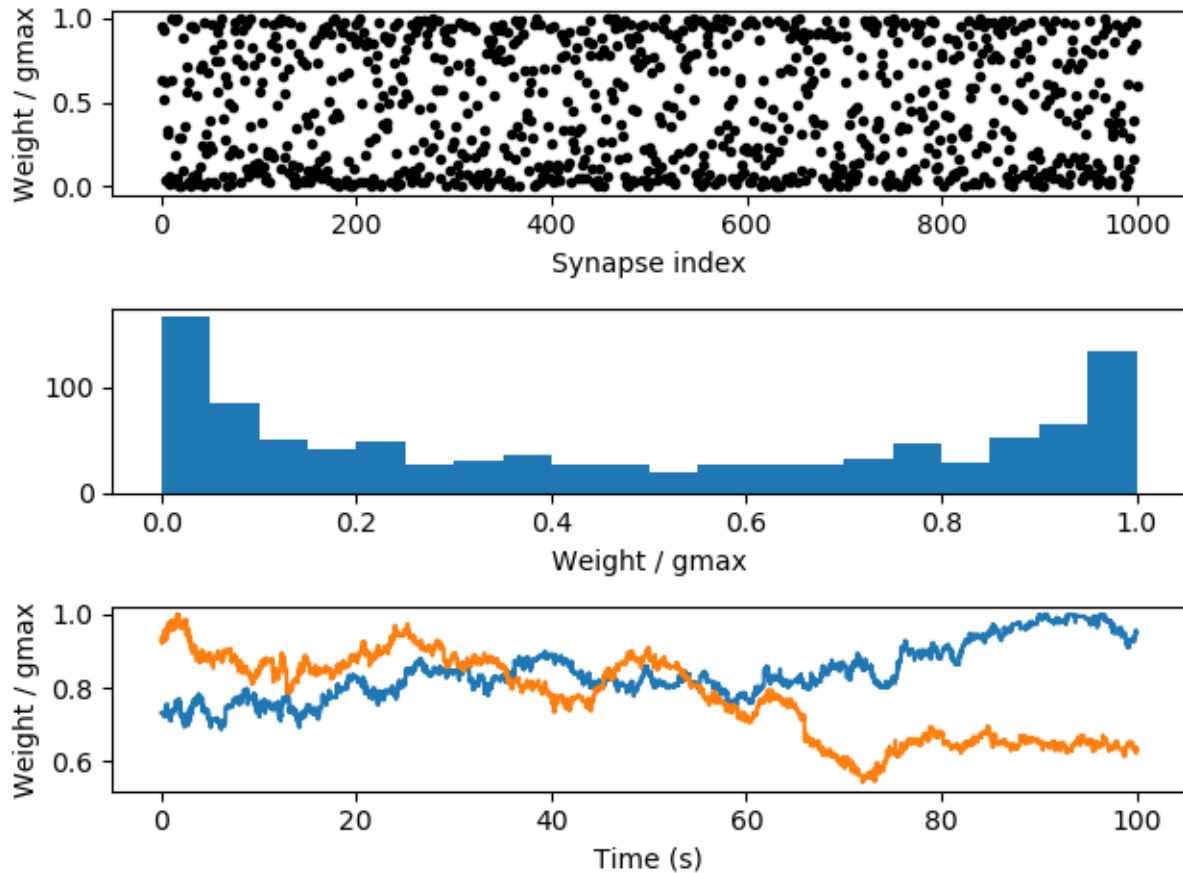
5.15.2 Example: cuba_openmp

Run the `cuba.py` example with OpenMP threads.

```python
from brian2 import *

set_device('cpp_standalone', directory='CUBA')
prefs.devices.cpp_standalone.openmp_threads = 4

taum = 20*ms
taue = 5*ms
taui = 10*ms
Vt = -50*mV
Vr = -60*mV
El = -49*mV


eqs = '''
dv/dt  = (ge+gi-(v-El))/taum : volt (unless refractory)
dge/dt = -ge/taue : volt (unless refractory)
dgi/dt = -gi/taui : volt (unless refractory)
'''
```

(continues on next page)

```
P = NeuronGroup(4000, eqs, threshold='v>Vt', reset='v = Vr', refractory=5*ms,
                method='exact')
P.v = 'Vr + rand() * (Vt - Vr)'
P.ge = 0*mV
P.gi = 0*mV

we = (60*0.27/10)*mV # excitatory synaptic weight (voltage)
wi = (-20*4.5/10)*mV # inhibitory synaptic weight
Ce = Synapses(P, P, on_pre='ge += we')
Ci = Synapses(P, P, on_pre='gi += wi')
Ce.connect('i<3200', p=0.02)
Ci.connect('i>=3200', p=0.02)

s_mon = SpikeMonitor(P)

run(1 * second)

plot(s_mon.t/ms, s_mon.i, ',k')
xlabel('Time (ms)')
ylabel('Neuron index')
show()
```
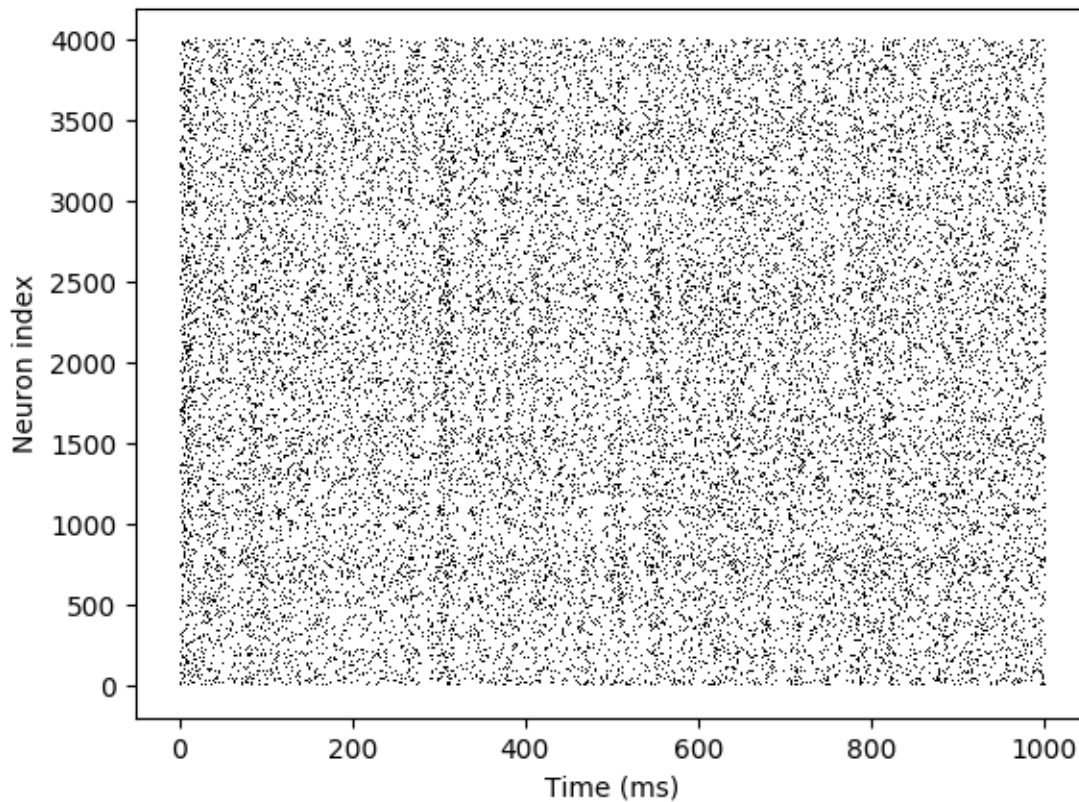
5.15.3 Example: simple_case

The most simple case how to use standalone mode.

```python
from brian2 import *
set_device('cpp_standalone')  # ← only difference to "normal" simulation

tau = 10*ms
eqs = '''
dv/dt = (1-v)/tau : 1
'''
G = NeuronGroup(10, eqs, method='exact')
G.v = 'rand()'
mon = StateMonitor(G, 'v', record=True)
run(100*ms)

plt.plot(mon.t/ms, mon.v.T)
plt.gca().set(xlabel='t (ms)', ylabel='v')
plt.show()
```
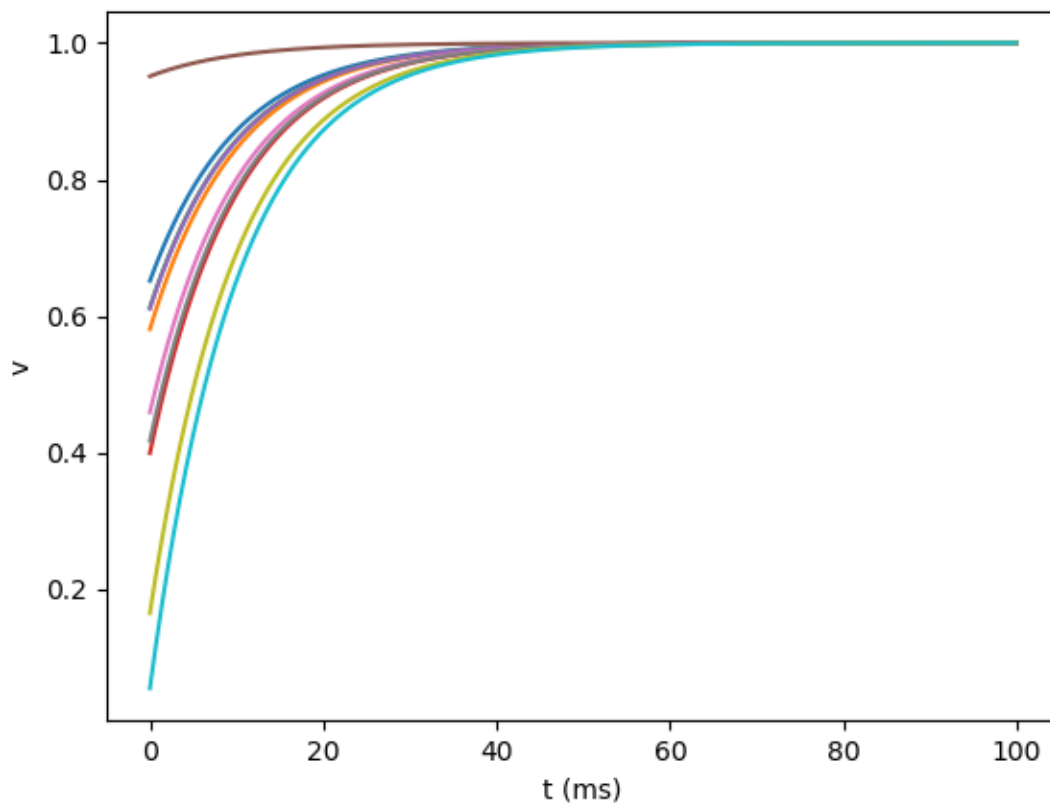
5.15.4 Example: simple_case_build

The most simple case how to use standalone mode with several *run()* calls.

```python
from brian2 import *
set_device('cpp_standalone', build_on_run=False)

tau = 10*ms
I = 1  # input current
eqs = '''
dv/dt = (I-v)/tau : 1
'''
G = NeuronGroup(10, eqs, method='exact')
G.v = 'rand()'
mon = StateMonitor(G, 'v', record=True)
run(20*ms)
I = 0
run(80*ms)
# Actually generate/compile/run the code:
device.build()

plt.plot(mon.t/ms, mon.v.T)
plt.gca().set(xlabel='t (ms)', ylabel='v')
plt.show()
```
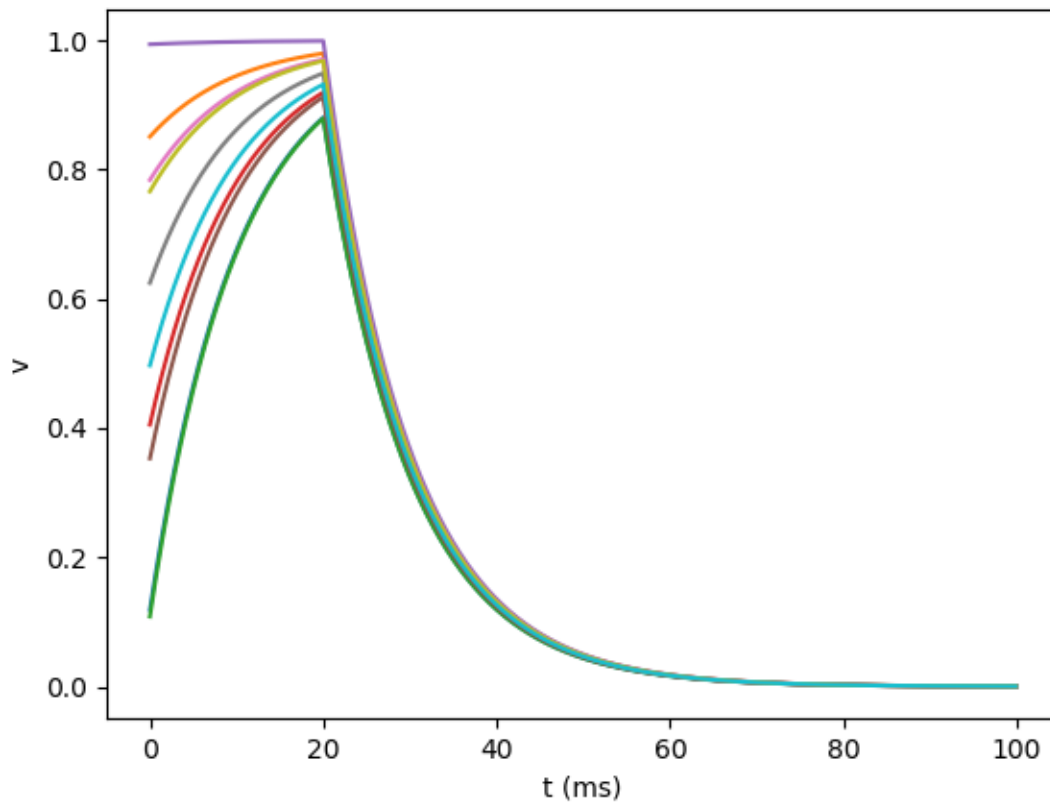
5.15.5 Example: standalone_multiplerun

This example shows how to run several, independent simulations in standalone mode. Note that this is not the optimal approach if running the same model with minor differences (as in this example).

The example come from Tutorial part 3. For a discussion see this post on the Brian forum.

```python
import numpy as np
import pylab as plt
import brian2 as b2
from time import time

b2.set_device('cpp_standalone')


def simulate(tau):
    # These two lines are needed to start a new standalone simulation:
    b2.device.reinit()
    b2.device.activate()

    eqs = '''
    dv/dt = -v/tau : 1
    '''

    net = b2.Network()
    P = b2.PoissonGroup(num_inputs, rates=input_rate)
    G = b2.NeuronGroup(1, eqs, threshold='v>1', reset='v=0', method='euler')
    S = b2.Synapses(P, G, on_pre='v += weight')
    S.connect()
    M = b2.SpikeMonitor(G)
    net.add([P, G, S, M])

    net.run(1000 * b2.ms)

    return M


if __name__ == "__main__":
    start_time = time()
    num_inputs = 100
    input_rate = 10 * b2.Hz
    weight = 0.1
    npoints = 15
    tau_range = np.linspace(1, 15, npoints) * b2.ms

    output_rates = np.zeros(npoints)
    for ii in range(npoints):
        tau_i = tau_range[ii]
        M = simulate(tau_i)
        output_rates[ii] = M.num_spikes / b2.second

    print(f"Done in {time() - start_time}")

    plt.plot(tau_range/b2.ms, output_rates)
    plt.xlabel(r"$\tau$ (ms)")
    plt.ylabel("Firing rate (sp/s)")
    plt.show()
```
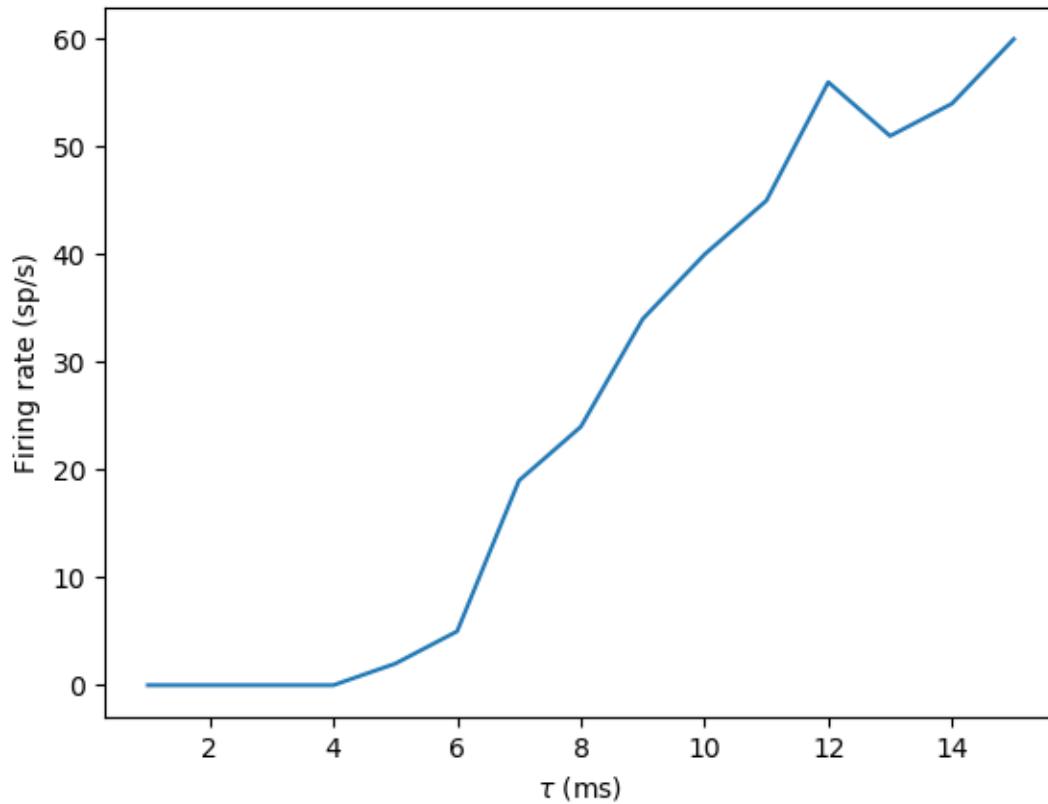
5.16 synapses

5.16.1 Example: STDP

Spike-timing dependent plasticity

Adapted from Song, Miller and Abbott (2000) and Song and Abbott (2001)

```python
from brian2 import *

N = 1000
taum = 10*ms
taupre = 20*ms
taupost = taupre
Ee = 0*mV
vt = -54*mV
vr = -60*mV
El = -74*mV
taue = 5*ms
F = 15*Hz
gmax = .01
```

```
dApre = .01
dApost = -dApre * taupre / taupost * 1.05
dApost *= gmax
dApre *= gmax

eqs_neurons = '''
dv/dt = (ge * (Ee-v) + El - v) / taum : volt
dge/dt = -ge / taue : 1
'''

poisson_input = PoissonGroup(N, rates=F)
neurons = NeuronGroup(1, eqs_neurons, threshold='v>vt', reset='v = vr',
                      method='euler')
S = Synapses(poisson_input, neurons,
             '''w : 1
                dApre/dt = -Apre / taupre : 1 (event-driven)
                dApost/dt = -Apost / taupost : 1 (event-driven)''',
             on_pre='''ge += w
                    Apre += dApre
                    w = clip(w + Apost, 0, gmax)''',
             on_post='''Apost += dApost
                     w = clip(w + Apre, 0, gmax)''',
             )
S.connect()
S.w = 'rand() * gmax'
mon = StateMonitor(S, 'w', record=[0, 1])
s_mon = SpikeMonitor(poisson_input)

run(100*second, report='text')

subplot(311)
plot(S.w / gmax, '.k')
ylabel('Weight / gmax')
xlabel('Synapse index')
subplot(312)
hist(S.w / gmax, 20)
xlabel('Weight / gmax')
subplot(313)
plot(mon.t/second, mon.w.T/gmax)
xlabel('Time (s)')
ylabel('Weight / gmax')
tight_layout()
show()
```
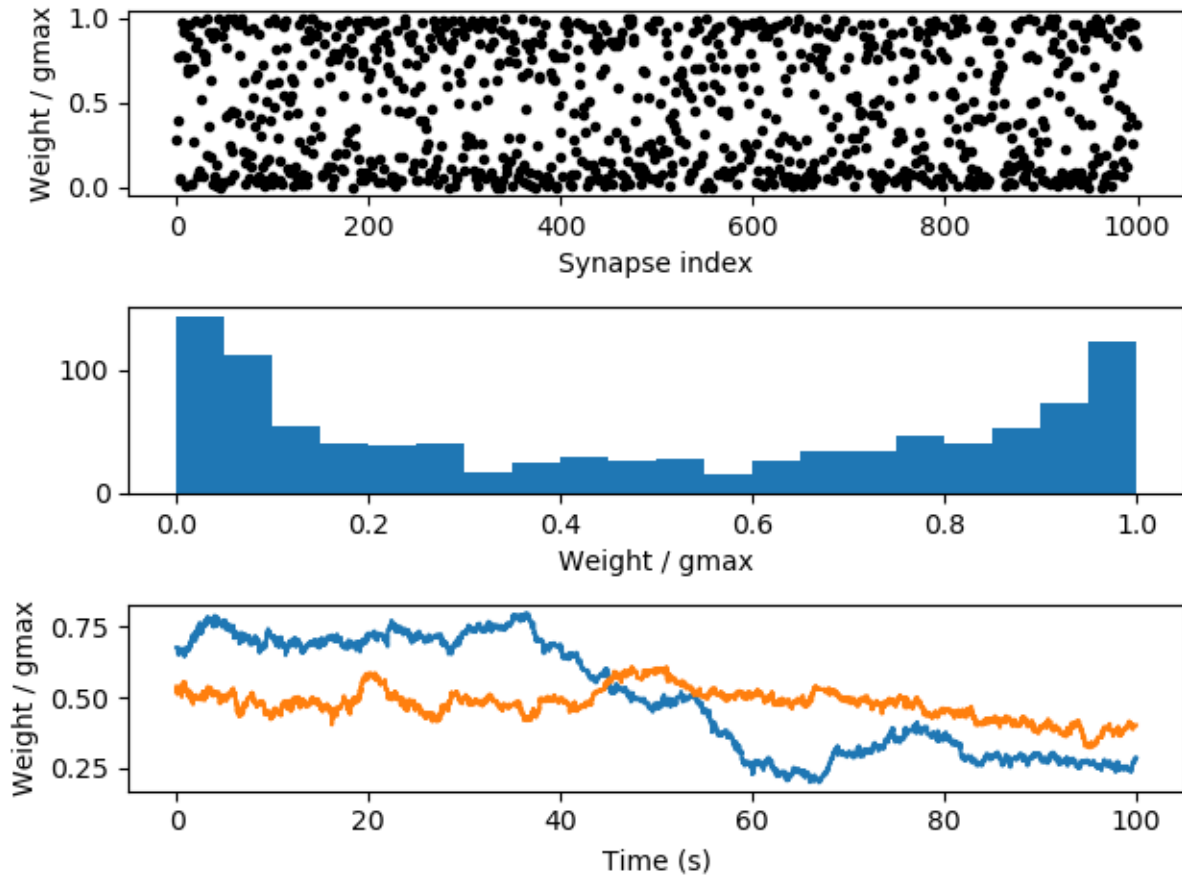
5.16.2 Example: continuous_interaction

Synaptic model with continuous interaction

This example implements a conductance base synapse that is continuously linking two neurons, i.e. the synaptic gating variable updates at each time step. Two Reduced Traub-Miles Model (RTM) neurons are connected to each other through a directed synapse from neuron 1 to 2.

Here, the complexity stems from the fact that the synaptic conductance is a continuous function of the membrane potential, instead of being triggered by individual spikes. This can be useful in particular when analyzing models mathematically but it is not recommended in most cases because they tend to be less efficient. Also note that this model only works with (pre-synaptic) neuron models that model the action potential in detail, i.e. not with integrate-and-fire type models.

There are two broad approaches (s as part of the pre-synaptic neuron or s as part of the Synapses object), all depends on whether the time constants are the same across all synapses or whether they can vary between synapses. In this example, the time constant is assumed to be the same and s is therefore part of the pre-synaptic neuron model.

References:

- Introduction to modeling neural dynamics, Börgers, chapter 20

- Discussion in Brian forum

```python
from brian2 import *

I_e = 1.5*uA
simulation_time = 100*ms
# neuron RTM parameters
El = -67 * mV
EK = -100 * mV
ENa = 50 * mV
ESyn = 0 * mV
gl = 0.1 * msiemens
gK = 80 * msiemens
gNa = 100 * msiemens


C = 1 * ufarad


weight = 0.25
gSyn = 1.0 * msiemens
tau_d = 2 * ms
tau_r = 0.2 * ms


# forming RTM model with differential equations
eqs = """
alphah = 0.128 * exp(-(vm + 50.0*mV) / (18.0*mV))/ms :Hz
alpham = 0.32/mV * (vm + 54*mV) / (1.0 - exp(-(vm + 54.0*mV) / (4.0*mV)))/ms:Hz
alphan = 0.032/mV * (vm + 52*mV) / (1.0 - exp(-(vm + 52.0*mV) / (5.0*mV)))/ms:Hz


betah  = 4.0 / (1.0 + exp(-(vm + 27.0*mV) / (5.0*mV)))/ms:Hz
betam  = 0.28/mV * (vm + 27.0*mV) / (exp((vm + 27.0*mV) / (5.0*mV)) - 1.0)/ms:Hz
betan  = 0.5 * exp(-(vm + 57.0*mV) / (40.0*mV))/ms:Hz


membrane_Im = I_ext + gNa*m**3*h*(ENa-vm) +
            gl*(El-vm) + gK*n**4*(EK-vm) + gSyn*s_in*(-vm): amp
I_ext : amp
s_in  : 1


dm/dt = alpham*(1-m)-betam*m : 1
dn/dt = alphan*(1-n)-betan*n : 1
dh/dt = alphah*(1-h)-betah*h : 1


ds/dt = 0.5 * (1 + tanh(0.1*vm/mV)) * (1-s)/tau_r - s/tau_d : 1


dvm/dt = membrane_Im/C : volt
"""


neuron = NeuronGroup(2, eqs, method="exponential_euler")

# initialize variables
neuron.vm = [-70.0, -65.0]*mV
neuron.m = "alpham / (alpham + betam)"
neuron.h = "alphah / (alphah + betah)"
neuron.n = "alphan / (alphan + betan)"
neuron.I_ext = [I_e, 0.0*uA]

S = Synapses(neuron,
            neuron,
            's_in_post = weight*s_pre:1 (summed)')
S.connect(i=0, j=1)
```

```
# tracking variables
st_mon = StateMonitor(neuron, ["vm", "s", "s_in"], record=[0, 1])

# running the simulation
run(simulation_time)

# plot the results
fig, ax = plt.subplots(2, figsize=(10, 6), sharex=True,
                       gridspec_kw={'height_ratios': (3, 1)})

ax[0].plot(st_mon.t/ms, st_mon.vm[0]/mV,
           lw=2, c="r", alpha=0.5, label="neuron 0")
ax[0].plot(st_mon.t/ms, st_mon.vm[1]/mV,
           lw=2, c="b", alpha=0.5, label='neuron 1')
ax[1].plot(st_mon.t/ms, st_mon.s[0],
           lw=2, c="r", alpha=0.5, label='s, neuron 0')
ax[1].plot(st_mon.t/ms, st_mon.s_in[1],
           lw=2, c="b", alpha=0.5, label='s_in, neuron 1')
ax[0].set(ylabel='v [mV]', xlim=(0, np.max(st_mon.t / ms)),
          ylim=(-100, 50))
ax[1].set(xlabel="t [ms]", ylabel="s", ylim=(0, 1))

ax[0].legend()
ax[1].legend()

plt.show()
```
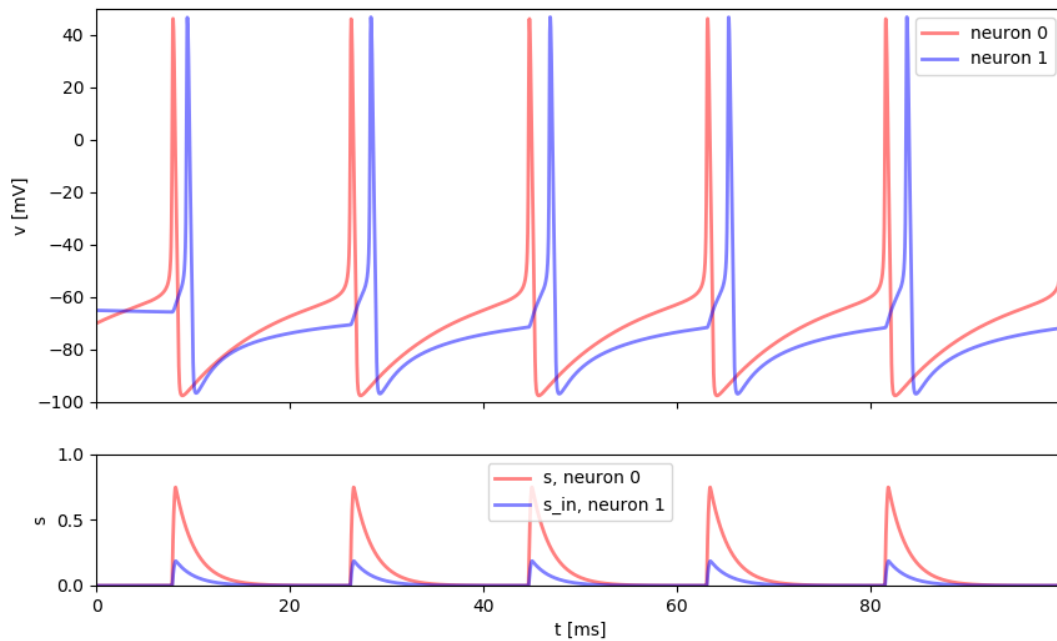
5.16.3 Example: efficient_gaussian_connectivity

An example of turning an expensive *Synapses.connect* operation into three cheap ones using a mathematical trick.

Consider the connection probability between neurons i and j given by the Gaussian function $p = e^{-\alpha(i-j)^2}$ (for some constant α). If we want to connect neurons with this probability, we can very simply do:

```
S.connect(p='exp(-alpha*(i-j)**2)')
```

However, this has a problem. Although we know that this will create $O(N)$ synapses if N is the number of neurons, because we have specified p as a function of i and j, we have to evaluate p(i, j) for every pair (i, j), and therefore it takes $O(N^2)$ operations.

Our first option is to take a cutoff, and say that if $p < q$ for some small q, then we assume that $p \approx 0$. We can work out which j values are compatible with a given value of i by solving $e^{-\alpha(i-j)^2} < q$ which gives $|i-j| < \sqrt{-\log(q)/\alpha} = w$. Now we implement the rule using the generator syntax to only search for values between i-w and i+w, except that some of these values will be outside the valid range of values for j so we set `skip_if_invalid=True`. The connection code is then:

```
S.connect(j='k for k in range(i-w, i+w) if rand()<exp(-alpha*(i-j)**2)',
          skip_if_invalid=True)
```

This is a lot faster (see graph labelled "Limited" for this algorithm).

However, it may be a problem that we have to specify a cutoff and so we will lose some synapses doing this: it won't be mathematically exact. This isn't a problem for the Gaussian because w grows very slowly with the cutoff probability q, but for other probability distributions with more weight in the tails, it could be an issue.

If we want to be exact, we can still do a big improvement. For the case $i - w \le j \le i + w$ we use the same connection code, but we also handle the case $|i - j| > w$. This time, we note that we want to create a synapse with probability $p(i - j)$ and we can rewrite this as $p(i - j)/p(w) \cdot p(w)$. If $|i - j| > w$ then this is a product of two probabilities $p(i - j)/p(w)$ and $p(w)$. So in the region $|i - j| > w$ a synapse will be created if two random events both occur, with these two probabilities. This might seem a little strange until you notice that one of the two probabilities $p(w)$ doesn't depend on i or j. This lets us use the much more efficient `sample` algorithm to generate a set of candidate j values, and then add the additional test `rand()<p(i-j)/p(w)`. Here's the code for that:

```
w = int(ceil(sqrt(log(q)/-0.1)))
S.connect(j='k for k in range(i-w, i+w) if rand()<exp(-alpha*(i-j)**2)',
          skip_if_invalid=True)
pmax = exp(-0.1*w**2)
S.connect(j='k for k in sample(0, i-w, p=pmax) if rand()<exp(-alpha*(i-j)**2)/pmax',
          skip_if_invalid=True)
S.connect(j='k for k in sample(i+w, N_post, p=pmax) if rand()<exp(-alpha*(i-j)**2)/
→pmax',
          skip_if_invalid=True)
```

This "Divided" method is also much faster than the naive method, and is mathematically correct. Note though that this method is still $O(N^2)$ but the constants are much, much smaller and this will usually be sufficient. It is possible to take the ideas developed here even further and get even better scaling, but in most cases it's unlikely to be worth the effort.

The code below shows these examples written out, along with some timing code and plots for different values of N.

```
from brian2 import *
import time

def naive(N):
```

```python
    G = NeuronGroup(N, 'v:1', threshold='v>1', name='G')
    S = Synapses(G, G, on_pre='v += 1', name='S')
    S.connect(p='exp(-0.1*(i-j)**2)')

def limited(N, q=0.001):
    G = NeuronGroup(N, 'v:1', threshold='v>1', name='G')
    S = Synapses(G, G, on_pre='v += 1', name='S')
    w = int(ceil(sqrt(log(q)/-0.1)))
    S.connect(j='k for k in range(i-w, i+w) if rand()<exp(-0.1*(i-j)**2)', skip_if_
→invalid=True)

def divided(N, q=0.001):
    G = NeuronGroup(N, 'v:1', threshold='v>1', name='G')
    S = Synapses(G, G, on_pre='v += 1', name='S')
    w = int(ceil(sqrt(log(q)/-0.1)))
    S.connect(j='k for k in range(i-w, i+w) if rand()<exp(-0.1*(i-j)**2)', skip_if_
→invalid=True)
    pmax = exp(-0.1*w**2)
    S.connect(j='k for k in sample(0, i-w, p=pmax) if rand()<exp(-0.1*(i-j)**2)/pmax',
→ skip_if_invalid=True)
    S.connect(j='k for k in sample(i+w, N_post, p=pmax) if rand()<exp(-0.1*(i-j)**2)/
→pmax', skip_if_invalid=True)

def repeated_run(f, N, repeats):
    start_time = time.time()
    for _ in range(repeats):
        f(N)
    end_time = time.time()
    return (end_time-start_time)/repeats

N = array([100, 500, 1000, 5000, 10000, 20000])
repeats = array([100, 10, 10, 1, 1, 1])*3
naive(10)
limited(10)
divided(10)
print('Starting naive')
loglog(N, [repeated_run(naive, n, r) for n, r in zip(N, repeats)],
       label='Naive', lw=2)
print('Starting limit')
loglog(N, [repeated_run(limited, n, r) for n, r in zip(N, repeats)],
       label='Limited', lw=2)
print('Starting divided')
loglog(N, [repeated_run(divided, n, r) for n, r in zip(N, repeats)],
       label='Divided', lw=2)
xlabel('N')
ylabel('Time (s)')
legend(loc='best', frameon=False)
show()
```
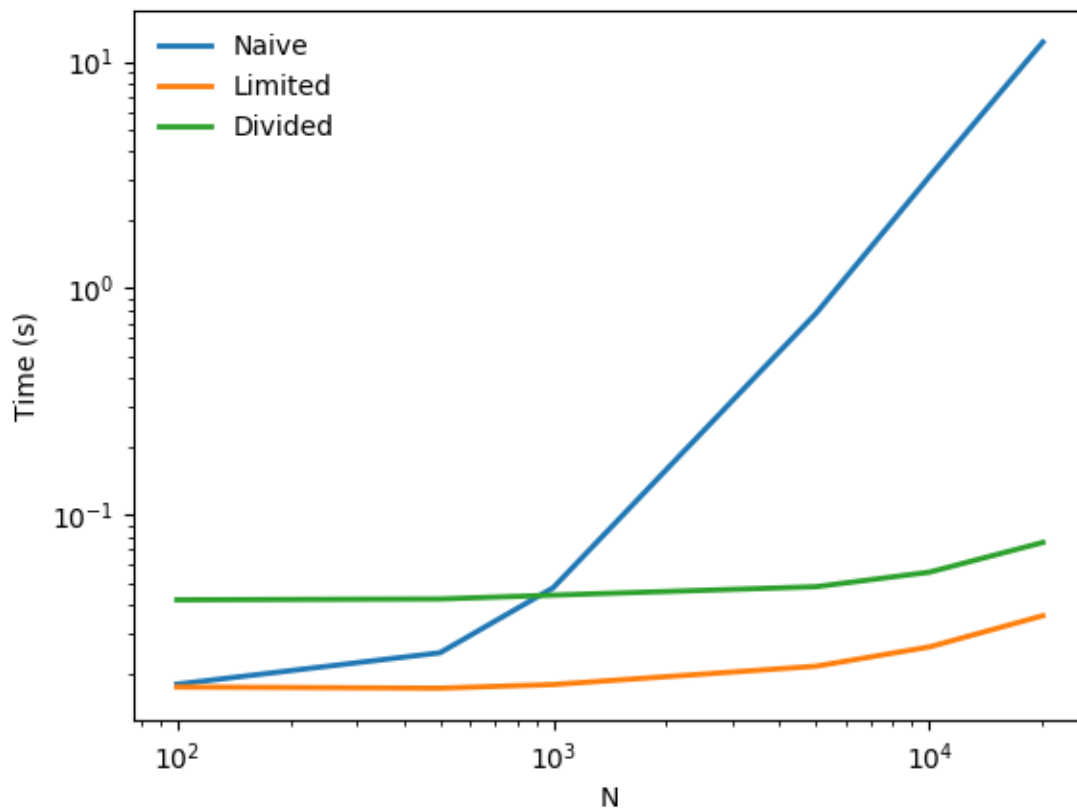
5.16.4 Example: gapjunctions

Neurons with gap junctions.

```python
from brian2 import *

n = 10
v0 = 1.05
tau = 10*ms

eqs = '''
dv/dt = (v0 - v + Igap) / tau : 1
Igap : 1 # gap junction current
'''

neurons = NeuronGroup(n, eqs, threshold='v > 1', reset='v = 0',
                      method='exact')
neurons.v = 'i * 1.0 / (n-1)'
trace = StateMonitor(neurons, 'v', record=[0, 5])

S = Synapses(neurons, neurons, '''
             w : 1 # gap junction conductance
```

```
            Igap_post = w * (v_pre - v_post) : 1 (summed)
            ''')
S.connect()
S.w = .02

run(500*ms)

plot(trace.t/ms, trace[0].v)
plot(trace.t/ms, trace[5].v)
xlabel('Time (ms)')
ylabel('v')
show()
```
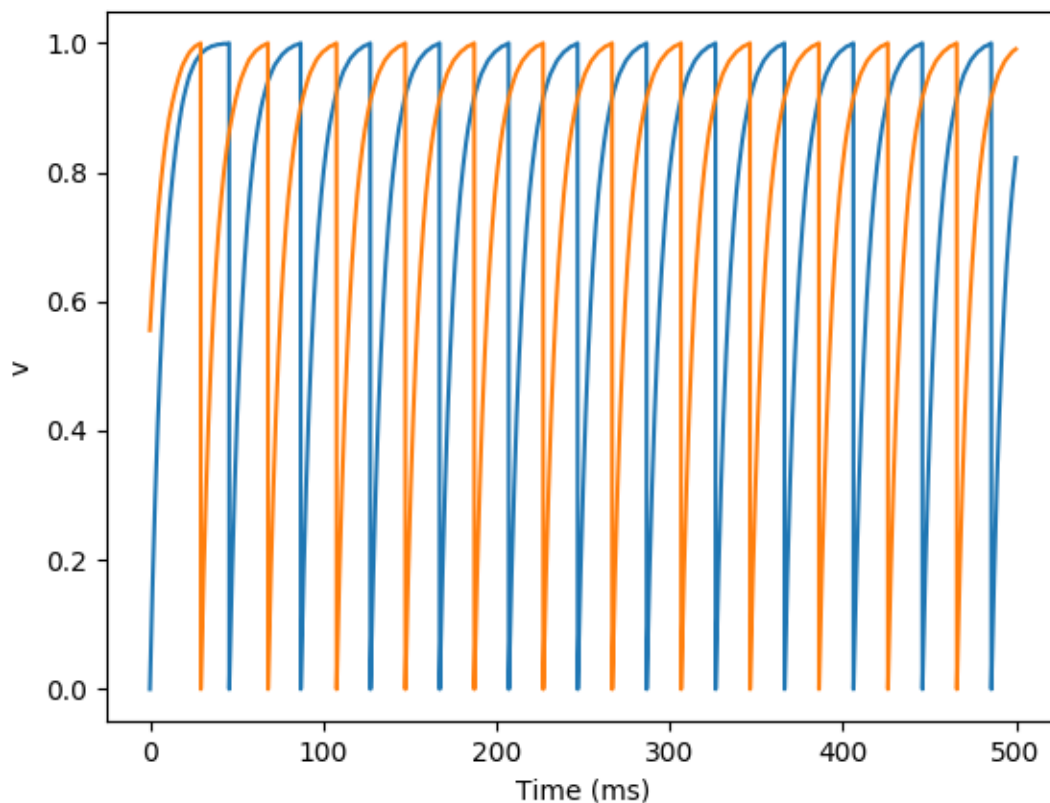

5.16.5 Example: jeffress

Jeffress model, adapted with spiking neuron models. A sound source (white noise) is moving around the head. Delay differences between the two ears are used to determine the azimuth of the source. Delays are mapped to a neural place code using delay lines (each neuron receives input from both ears, with different delays).

```
from brian2 import *
```

```python
defaultclock.dt = .02*ms

# Sound
sound = TimedArray(10 * randn(50000), dt=defaultclock.dt) # white noise

# Ears and sound motion around the head (constant angular speed)
sound_speed = 300*metre/second
interaural_distance = 20*cm # big head!
max_delay = interaural_distance / sound_speed
print("Maximum interaural delay: %s" % max_delay)
angular_speed = 2 * pi / second # 1 turn/second
tau_ear = 1*ms
sigma_ear = .1
eqs_ears = '''
dx/dt = (sound(t-delay)-x)/tau_ear+sigma_ear*(2./tau_ear)**.5*xi : 1 (unless
→refractory)
delay = distance*sin(theta) : second
distance : second # distance to the centre of the head in time units
dtheta/dt = angular_speed : radian
'''
ears = NeuronGroup(2, eqs_ears, threshold='x>1', reset='x = 0',
                   refractory=2.5*ms, name='ears', method='euler')
ears.distance = [-.5 * max_delay, .5 * max_delay]
traces = StateMonitor(ears, 'delay', record=True)
# Coincidence detectors
num_neurons = 30
tau = 1*ms
sigma = .1
eqs_neurons = '''
dv/dt = -v / tau + sigma * (2 / tau)**.5 * xi : 1
'''
neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1',
                      reset='v = 0', name='neurons', method='euler')

synapses = Synapses(ears, neurons, on_pre='v += .5')
synapses.connect()

synapses.delay['i==0'] = '(1.0*j)/(num_neurons-1)*1.1*max_delay'
synapses.delay['i==1'] = '(1.0*(num_neurons-j-1))/(num_neurons-1)*1.1*max_delay'

spikes = SpikeMonitor(neurons)

run(1000*ms)

# Plot the results
i, t = spikes.it
subplot(2, 1, 1)
plot(t/ms, i, '.')
xlabel('Time (ms)')
ylabel('Neuron index')
xlim(0, 1000)
subplot(2, 1, 2)
plot(traces.t/ms, traces.delay.T/ms)
xlabel('Time (ms)')
ylabel('Input delay (ms)')
xlim(0, 1000)
tight_layout()
```

```
show()
```
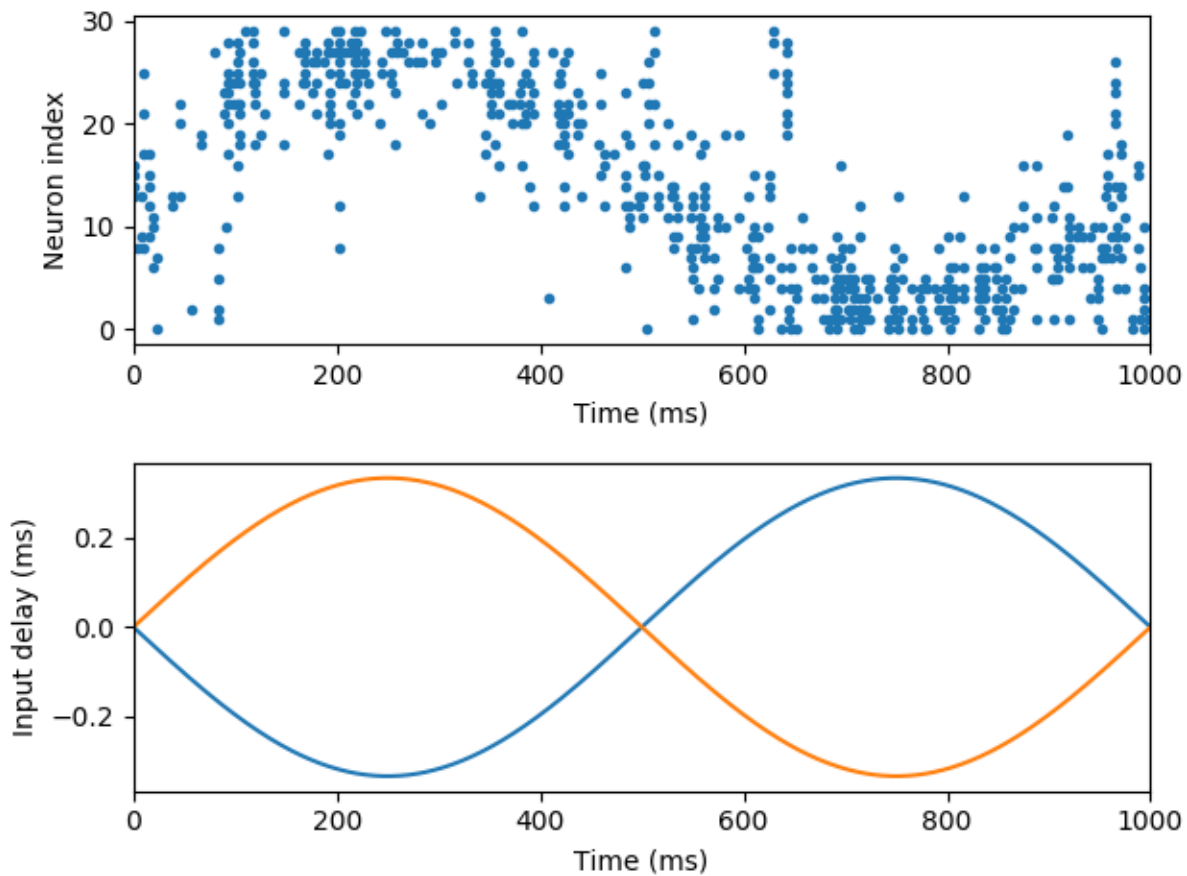

5.16.6 Example: licklider

Spike-based adaptation of Licklider's model of pitch processing (autocorrelation with delay lines) with phase locking.

```python
from brian2 import *

defaultclock.dt = .02 * ms

# Ear and sound
max_delay = 20*ms # 50 Hz
tau_ear = 1*ms
sigma_ear = 0.0
eqs_ear = '''
dx/dt = (sound-x)/tau_ear+0.1*(2./tau_ear)**.5*xi : 1 (unless refractory)
sound = 5*sin(2*pi*frequency*t)**3 : 1 # nonlinear distortion
#sound = 5*(sin(4*pi*frequency*t)+.5*sin(6*pi*frequency*t)) : 1 # missing fundamental
frequency = (200+200*t*Hz)*Hz : Hz # increasing pitch
'''
receptors = NeuronGroup(2, eqs_ear, threshold='x>1', reset='x=0',
```

```
                                        refractory=2*ms, method='euler')
# Coincidence detectors
min_freq = 50*Hz
max_freq = 1000*Hz
num_neurons = 300
tau = 1*ms
sigma = .1
eqs_neurons = '''
dv/dt = -v/tau+sigma*(2./tau)**.5*xi : 1
'''

neurons = NeuronGroup(num_neurons, eqs_neurons, threshold='v>1', reset='v=0',
                      method='euler')

synapses = Synapses(receptors, neurons, on_pre='v += 0.5')
synapses.connect()
synapses.delay = 'i*1.0/exp(log(min_freq/Hz)+(j*1.0/(num_neurons-1))*log(max_freq/min_
→freq))*second'

spikes = SpikeMonitor(neurons)

run(500*ms)
plot(spikes.t/ms, spikes.i, '.k')
xlabel('Time (ms)')
ylabel('Frequency')
yticks([0, 99, 199, 299],
       array(1. / synapses.delay[1, [0, 99, 199, 299]], dtype=int))
show()
```
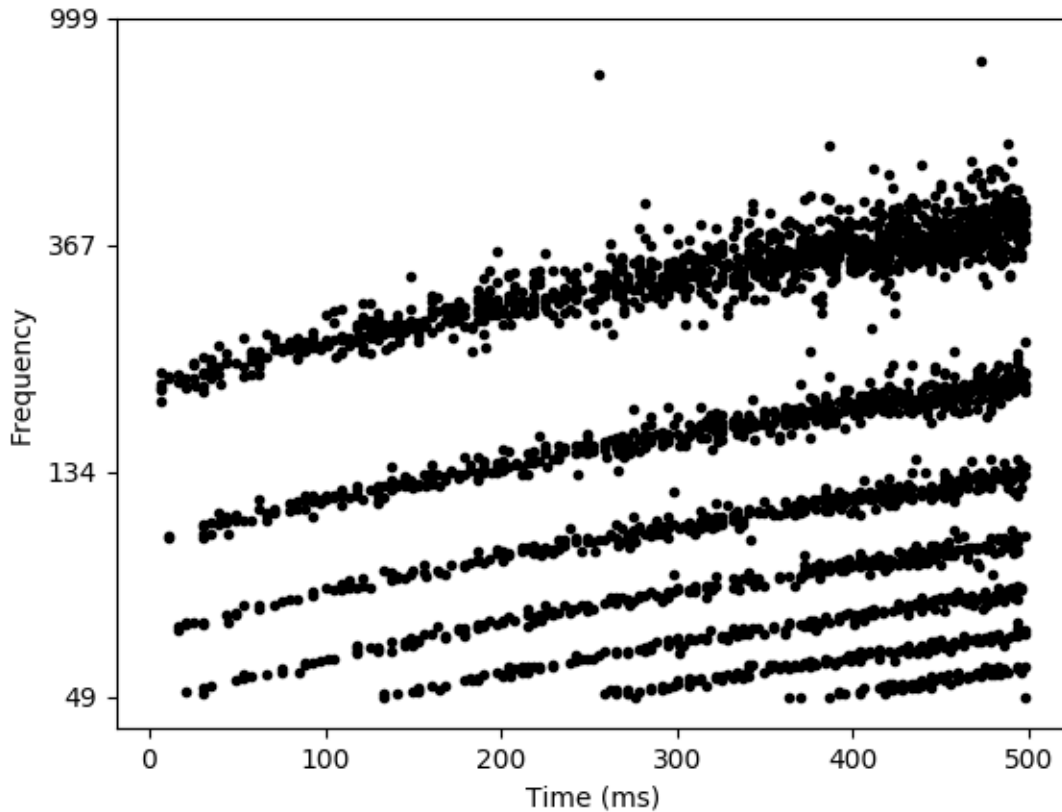
5.16.7 Example: nonlinear

NMDA synapses.

```python
from brian2 import *

a = 1 / (10*ms)
b = 1 / (10*ms)
c = 1 / (10*ms)

neuron_input = NeuronGroup(2, 'dv/dt = 1/(10*ms) : 1', threshold='v>1', reset='v = 0',
                    method='euler')
neurons = NeuronGroup(1, """dv/dt = (g-v)/(10*ms) : 1
                            g : 1""", method='exact')
S = Synapses(neuron_input, neurons, '''
                dg_syn/dt = -a*g_syn+b*x*(1-g_syn) : 1 (clock-driven)
                g_post = g_syn : 1 (summed)
                dx/dt=-c*x : 1 (clock-driven)
                w : 1 # synaptic weight
            ''', on_pre='x += w') # NMDA synapses

S.connect()
```

(continues on next page)

```
S.w = [1., 10.]
neuron_input.v = [0., 0.5]

M = StateMonitor(S, 'g',
                 # If not using standalone mode, this could also simply be
                 # record=True
                 record=np.arange(len(neuron_input)*len(neurons)))
Mn = StateMonitor(neurons, 'g', record=0)

run(1000*ms)

subplot(2, 1, 1)
plot(M.t/ms, M.g.T)
xlabel('Time (ms)')
ylabel('g_syn')
subplot(2, 1, 2)
plot(Mn.t/ms, Mn[0].g)
ylabel('Time (ms)')
ylabel('g')
tight_layout()
show()
```
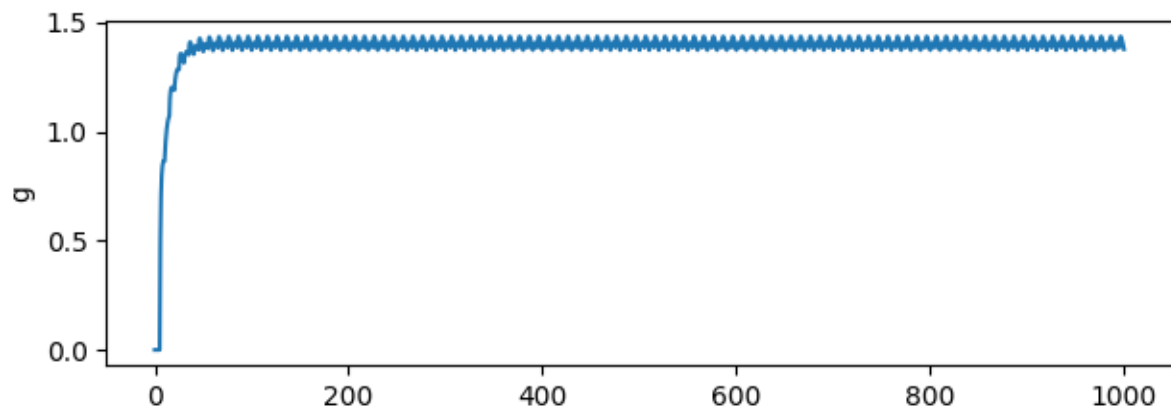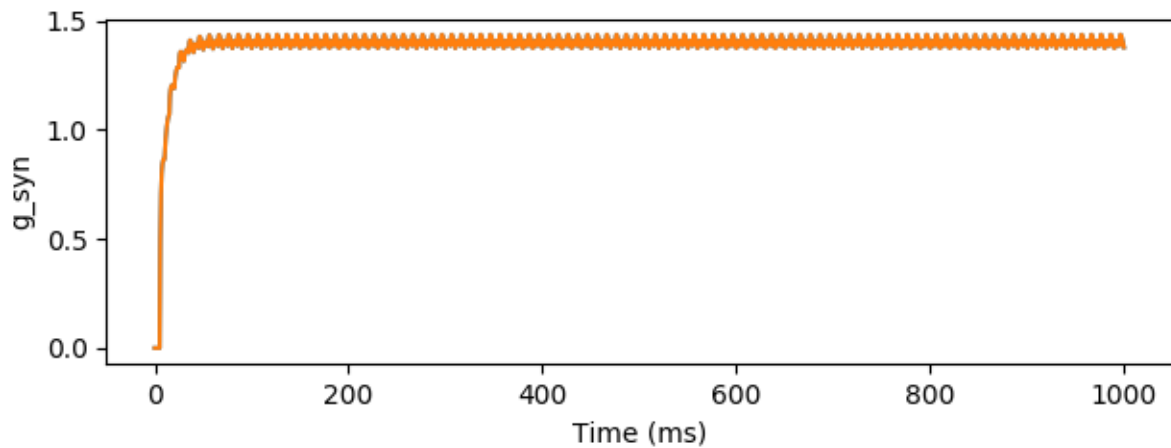
5.16.8 Example: spatial_connections

A simple example showing how string expressions can be used to implement spatial (deterministic or stochastic) connection patterns.

```python
from brian2 import *

rows, cols = 20, 20
G = NeuronGroup(rows * cols, '''x : meter
                                y : meter''')
# initialize the grid positions
grid_dist = 25*umeter
G.x = '(i // rows) * grid_dist - rows/2.0 * grid_dist'
G.y = '(i % rows) * grid_dist - cols/2.0 * grid_dist'

# Deterministic connections
distance = 120*umeter
S_deterministic = Synapses(G, G)
S_deterministic.connect('sqrt((x_pre - x_post)**2 + (y_pre - y_post)**2) < distance')

# Random connections (no self-connections)
S_stochastic = Synapses(G, G)
S_stochastic.connect('i != j',
                     p='1.5 * exp(-((x_pre-x_post)**2 + (y_pre-y_post)**2)/
↪(2*(60*umeter)**2))')

figure(figsize=(12, 6))

# Show the connections for some neurons in different colors
for color in ['g', 'b', 'm']:
    subplot(1, 2, 1)
    neuron_idx = np.random.randint(0, rows*cols)
    plot(G.x[neuron_idx] / umeter, G.y[neuron_idx] / umeter, 'o', mec=color,
        mfc='none')
    plot(G.x[S_deterministic.j[neuron_idx, :]] / umeter,
        G.y[S_deterministic.j[neuron_idx, :]] / umeter, color + '.')
    subplot(1, 2, 2)
    plot(G.x[neuron_idx] / umeter, G.y[neuron_idx] / umeter, 'o', mec=color,
        mfc='none')
    plot(G.x[S_stochastic.j[neuron_idx, :]] / umeter,
        G.y[S_stochastic.j[neuron_idx, :]] / umeter, color + '.')

for idx, t in enumerate(['determininstic connections',
                         'random connections']):
    subplot(1, 2, idx + 1)
    xlim((-rows/2.0 * grid_dist) / umeter, (rows/2.0 * grid_dist) / umeter)
    ylim((-cols/2.0 * grid_dist) / umeter, (cols/2.0 * grid_dist) / umeter)
    title(t)
    xlabel('x')
    ylabel('y', rotation='horizontal')
    axis('equal')

tight_layout()
show()
```
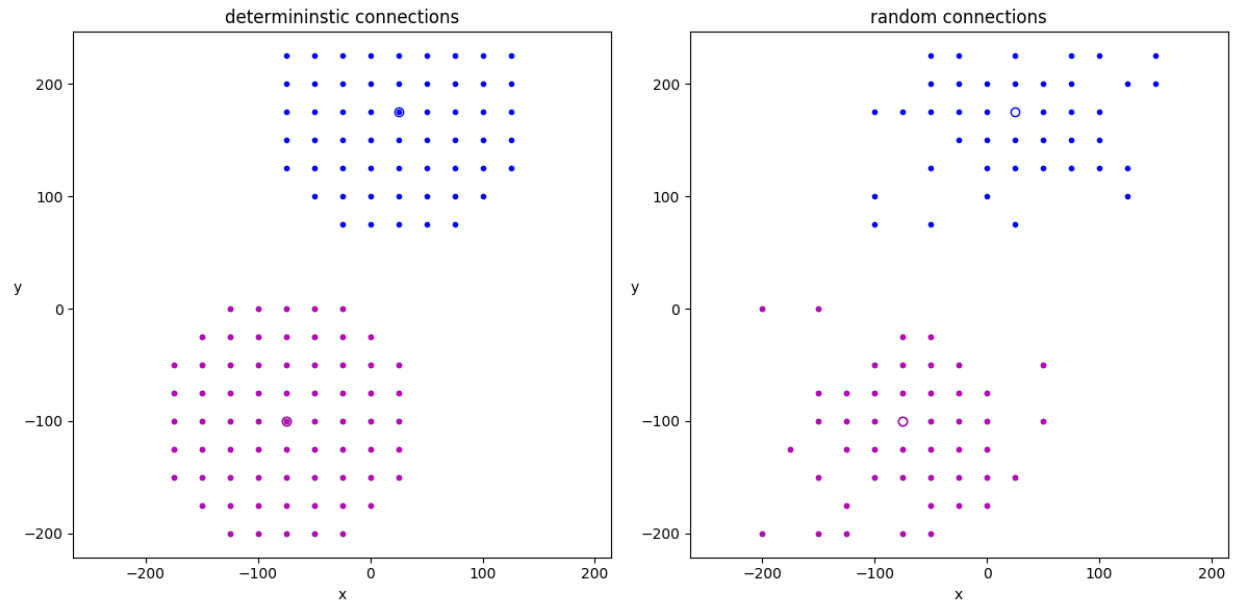
5.16.9 Example: spike_based_homeostasis

Following O. Breitwieser: "Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization"

Two poisson stimuli are connected to a neuron. One with a varying rate and the other with a fixed rate. The synaptic weight from the varying rate stimulus to the neuron is fixed. The synaptic weight from the fixed rate stimulus to the neuron is plastic and tries to keep the neuron at a firing rate that is determined by the parameters of the plasticity rule.

Sebastian Schmitt, 2021

```python
import itertools
import numpy as np
import matplotlib.pyplot as plt

from brian2 import TimedArray, PoissonGroup, NeuronGroup, Synapses, StateMonitor,
→PopulationRateMonitor
from brian2 import defaultclock, run
from brian2 import Hz, ms, second

# The synaptic weight from the steady stimulus is plastic
steady_stimulus = TimedArray([50]*Hz, dt=40*second)
steady_poisson = PoissonGroup(1, rates='steady_stimulus(t)')

# The synaptic weight from the varying stimulus is static
varying_stimulus = TimedArray([25*Hz, 50*Hz, 0*Hz, 35*Hz, 0*Hz], dt=10*second)
varying_poisson = PoissonGroup(1, rates='varying_stimulus(t)')

# dw_plus/dw_minus determines scales the steady stimulus rate to the target firing
→rate, must not be larger 1
# the magntude of dw_plus and dw_minus determines the "speed" of the homeostasis
parameters = {
    'tau': 10*ms,  # membrane time constant
    'dw_plus': 0.05,  # weight increment on pre spike
```

(continues on next page)

```python
    'dw_minus': 0.05,  # weight increment on post spike
    'w_max': 2,  # maximum plastic weight
    'w_initial': 0  # initial plastic weight
}

eqs = 'dv/dt = (0 - v)/tau : 1 (unless refractory)'

neuron_with_homeostasis = NeuronGroup(1, eqs,
                                      threshold='v > 1', reset='v = -1',
                                      method='euler', refractory=1*ms,
                                      namespace=parameters)
neuron_without_homeostasis = NeuronGroup(1, eqs,
                                         threshold='v > 1', reset='v = -1',
                                         method='euler', refractory=1*ms,
                                         namespace=parameters)

plastic_synapse = Synapses(steady_poisson, neuron_with_homeostasis,
                           'w : 1',
                           on_pre='''
v_post += w
w = clip(w + dw_plus, 0, w_max)
''',
                           on_post='''
w = clip(w - dw_minus, 0, w_max)
''', namespace=parameters)
plastic_synapse.connect()
plastic_synapse.w = parameters['w_initial']

non_plastic_synapse_neuron_without_homeostasis = Synapses(varying_poisson,
                                                          neuron_without_homeostasis,
                                                          'w : 1', on_pre='v_post += w
↪')
non_plastic_synapse_neuron_without_homeostasis.connect()
non_plastic_synapse_neuron_without_homeostasis.w = 2

non_plastic_synapse_neuron = Synapses(varying_poisson, neuron_with_homeostasis,
                                      'w : 1', on_pre='v_post += w')
non_plastic_synapse_neuron.connect()
non_plastic_synapse_neuron.w = 2

M = StateMonitor(neuron_with_homeostasis, 'v', record=True)
M2 = StateMonitor(plastic_synapse, 'w', record=True)
M_rate_neuron_with_homeostasis = PopulationRateMonitor(neuron_with_homeostasis)
M_rate_neuron_without_homeostasis = PopulationRateMonitor(neuron_without_homeostasis)

duration = 40*second
defaultclock.dt = 0.1*ms
run(duration, report='text')

fig, axes = plt.subplots(3, sharex=True)

axes[0].plot(M2.t/second, M2.w[0], label="homeostatic weight")
axes[0].set_ylabel("weight")
axes[0].legend()

# dt is in second
dts = np.arange(0., len(varying_stimulus.values)*varying_stimulus.dt, varying_
↪stimulus.dt)
```

```
x = list(itertools.chain(*zip(dts, dts)))
y = list(itertools.chain(*zip(varying_stimulus.values/Hz, varying_stimulus.values/
→Hz)))
axes[1].plot(x, [0] + y[:-1], label="varying stimulus")
axes[1].set_ylabel("rate [Hz]")
axes[1].legend()

# in ms
smooth_width = 100*ms
axes[2].plot(M_rate_neuron_with_homeostasis.t/second,
            M_rate_neuron_with_homeostasis.smooth_rate(width=smooth_width)/Hz,
            label="with homeostasis")
axes[2].plot(M_rate_neuron_without_homeostasis.t/second,
            M_rate_neuron_without_homeostasis.smooth_rate(width=smooth_width)/Hz,
            label="without homeostasis")
axes[2].set_ylabel("firing rate [Hz]")
axes[2].legend()

plt.xlabel('Time (s)')
plt.show()
```
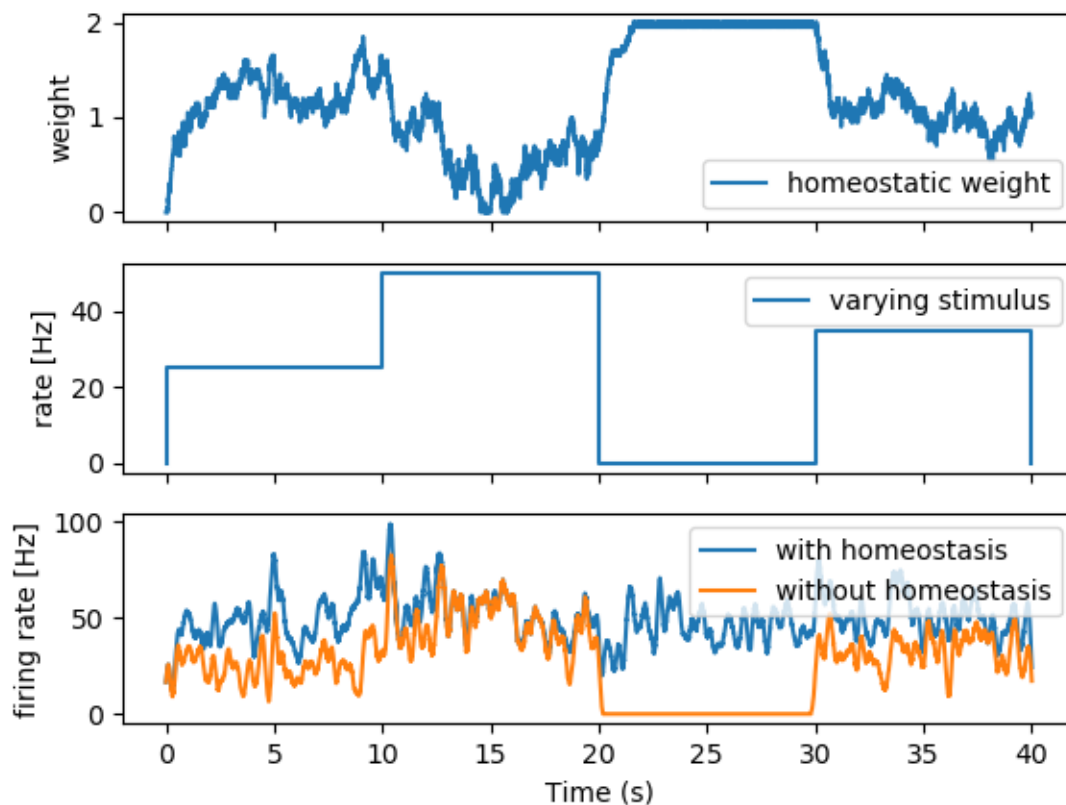
5.16.10 Example: state_variables

Set state variable values with a string (using code generation).

```python
from brian2 import *

G = NeuronGroup(100, 'v:volt', threshold='v>-50*mV')
G.v = '(sin(2*pi*i/N) - 70 + 0.25*randn()) * mV'
S = Synapses(G, G, 'w : volt', on_pre='v += w')
S.connect()

space_constant = 200.0
S.w['i > j'] = 'exp(-(i - j)**2/space_constant) * mV'

# Generate a matrix for display
w_matrix = np.zeros((len(G), len(G)))
w_matrix[S.i[:], S.j[:]] = S.w[:]

subplot(1, 2, 1)
plot(G.v[:] / mV)
xlabel('Neuron index')
ylabel('v')
subplot(1, 2, 2)
imshow(w_matrix)
xlabel('i')
ylabel('j')
title('Synaptic weight')
tight_layout()
show()
```
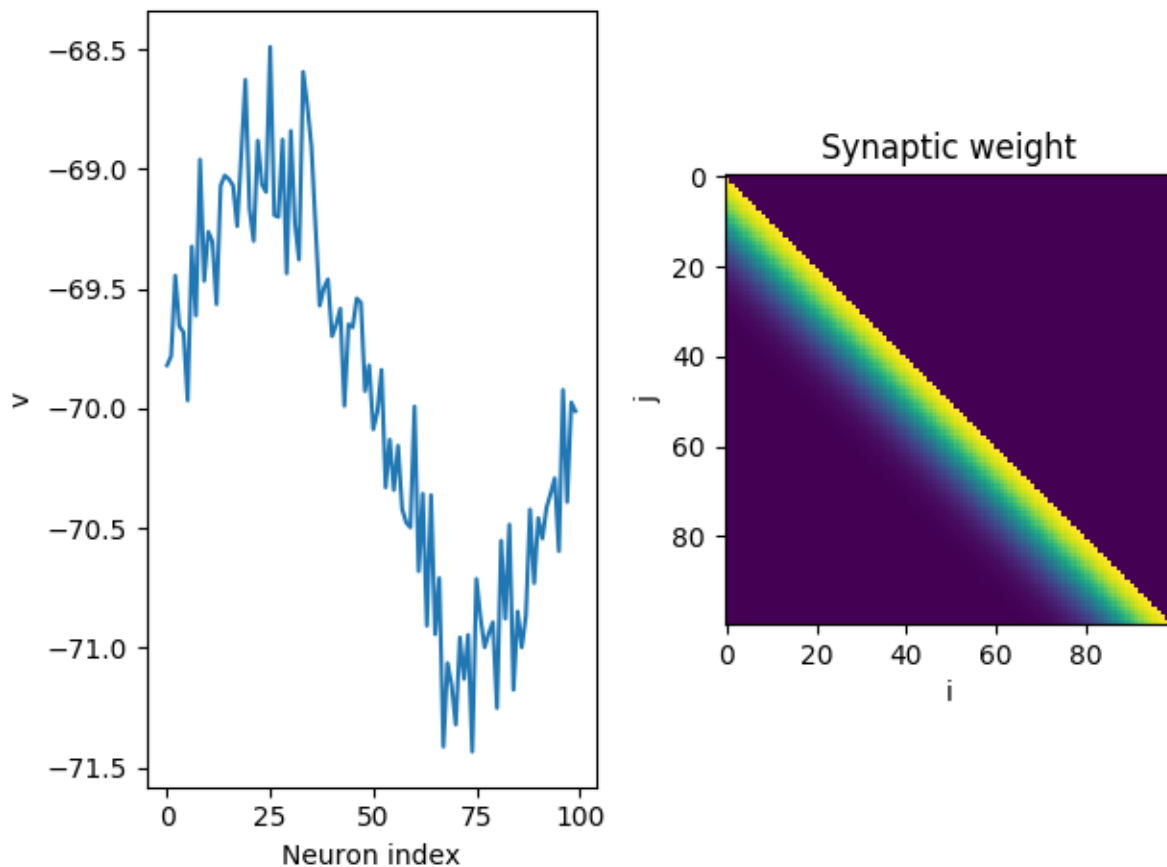
5.16.11 Example: synapses

A simple example of using *Synapses*.

```python
from brian2 import *

G1 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                 threshold='v > 1', reset='v=0.', method='exact')
G1.v = 1.2
G2 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : 1',
                 threshold='v > 1', reset='v=0', method='exact')

syn = Synapses(G1, G2, 'dw/dt = -w / (50*ms): 1 (event-driven)', on_pre='v += w')

syn.connect('i == j', p=0.75)

# Set the delays
syn.delay = '1*ms + i*ms + 0.25*ms * randn()'
# Set the initial values of the synaptic variable
syn.w = 1

mon = StateMonitor(G2, 'v', record=True)
```

(continues on next page)

```
run(20*ms)
plot(mon.t/ms, mon.v.T)
xlabel('Time (ms)')
ylabel('v')
show()
```
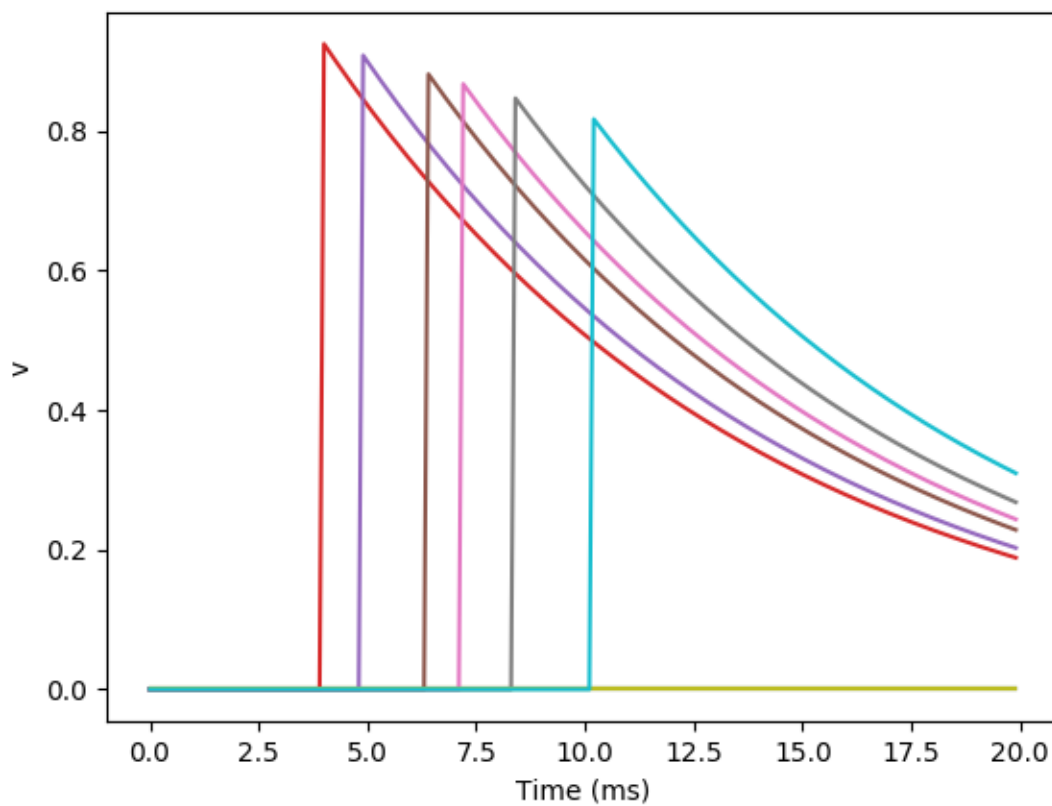
BRIAN2 PACKAGE

Brian 2

Functions

---

check_cache(target)	

6.1 check_cache function

(*Shortest import*: `from brian2.__init__ import check_cache`)

brian2.__init__.**check_cache**(*target*)

---

clear_cache(target)	Clears the on-disk cache with the compiled files for a given code generation target.

6.2 clear_cache function

(*Shortest import*: `from brian2.__init__ import clear_cache`)

brian2.__init__.**clear_cache**(*target*)
    Clears the on-disk cache with the compiled files for a given code generation target.

> **Parameters target** : str
>
> > The code generation target (e.g. `'cython'`)

Raises

ValueError If the given code generation target does not have an on-disk cache

IOError If the cache directory contains unexpected files, suggesting that deleting it would also delete files unrelated to the cache.

6.3 `_version` module

Functions

[*get_versions*](#)()

6.3.1 get_versions function

(*Shortest import*: `from brian2._version import get_versions`)

`brian2._version.`**`get_versions`**`()`

6.4 `hears` module

This is only a bridge for using Brian 1 hears with Brian 2.

Deprecated since version 2.2.2.2: Use the brian2hears package instead.

NOTES:

- Slicing sounds with Brian 2 units doesn't work, you need to either use Brian 1 units or replace calls to `sound[:20*ms]` with `sound.slice(None, 20*ms)`, etc.

TODO: handle properties (e.g. sound.duration)

Not working examples:

- time_varying_filter1 (care with units)

Exported members: *convert_unit_b1_to_b2*, *convert_unit_b2_to_b1*

Classes

[*BridgeSound*](#)()	We add a new method slice because slicing with units can't work with Brian 2 units.

6.4.1 BridgeSound class

(*Shortest import*: `from brian2.hears import BridgeSound`)

`class` `brian2.hears.`**`BridgeSound`**
    Bases: `brian2.hears.wrap_units_class.<locals>.new_class`

We add a new method slice because slicing with units can't work with Brian 2 units.

Methods

slice(*args)	

Details

slice(**args*)

FilterbankGroup(*args, **kw)	

Methods

6.4.2 FilterbankGroup class

(*Shortest import*: `from brian2.hears import FilterbankGroup`)

class brian2.hears.**FilterbankGroup**(**args*, ***kw*)
    Bases: *brian2.groups.neurongroup.NeuronGroup*

Methods

reinit()	

Details

reinit()

Sound	alias of *brian2.hears.BridgeSound*

6.4.3 Sound class

(*Shortest import*: `from brian2.hears import Sound`)

brian2.hears.**Sound**
    alias of *brian2.hears.BridgeSound*

WrappedSound	alias of brian2.hears.wrap_units_class. <locals>.new_class

6.4.4 WrappedSound class

(*Shortest import*: `from brian2.hears import WrappedSound`)

`brian2.hears.`**`WrappedSound`**
    alias of `brian2.hears.wrap_units_class.<locals>.new_class`

Functions

`convert_unit_b1_to_b2`(val)

6.4.5 convert_unit_b1_to_b2 function

(*Shortest import*: `from brian2.hears import convert_unit_b1_to_b2`)

`brian2.hears.`**`convert_unit_b1_to_b2`**(*val*)

`convert_unit_b2_to_b1`(val)

6.4.6 convert_unit_b2_to_b1 function

(*Shortest import*: `from brian2.hears import convert_unit_b2_to_b1`)

`brian2.hears.`**`convert_unit_b2_to_b1`**(*val*)

`modify_arg`(arg)	Modify arguments to make them compatible with Brian 1.

6.4.7 modify_arg function

(*Shortest import*: `from brian2.hears import modify_arg`)

`brian2.hears.`**`modify_arg`**(*arg*)
    Modify arguments to make them compatible with Brian 1.

- Arrays of units are replaced with straight arrays

- Single values are replaced with Brian 1 equivalents

- Slices are handled so we can use e.g. sound[:20*ms]

The second part was necessary because some functions/classes test if an object is an array or not to see if it is a sequence, but because brian2.Quantity derives from ndarray this was causing problems.

`wrap_units`(f)	Wrap a function to convert units into a form that Brian 1 can handle.

6.4.8 wrap_units function

(*Shortest import*: `from brian2.hears import wrap_units`)

`brian2.hears.`**`wrap_units`**(*f*)
> Wrap a function to convert units into a form that Brian 1 can handle. Also, check the output argument, if it is a `b1h.Sound` wrap it.

`wrap_units_class`(_C)	Wrap a class to convert units into a form that Brian 1 can handle in all methods

6.4.9 wrap_units_class function

(*Shortest import*: `from brian2.hears import wrap_units_class`)

`brian2.hears.`**`wrap_units_class`**(*_C*)
> Wrap a class to convert units into a form that Brian 1 can handle in all methods

`wrap_units_property`(p)	

6.4.10 wrap_units_property function

(*Shortest import*: `from brian2.hears import wrap_units_property`)

`brian2.hears.`**`wrap_units_property`**(*p*)

6.5 `numpy_` module

A dummy package to allow importing numpy and the unit-aware replacements of numpy functions without having to know which functions are overwritten.

This can be used for example as `import brian2.numpy_ as np`

Exported members: `ModuleDeprecationWarning`, `VisibleDeprecationWarning`, `__version__`, `show_config()`, `char`, `rec`, `memmap`, `newaxis`, `ndarray`, `flatiter`, `nditer`, `nested_iters`, `ufunc`, *`arange()`*, `array`, `zeros`, `count_nonzero()`, `empty`, `broadcast`, `dtype`, `fromstring`, `fromfile`, `frombuffer`, *`where()`*, `argwhere()` … (620 more members)

6.6 `only` module

A dummy package to allow wildcard import from brian2 without also importing the pylab (numpy + matplotlib) namespace.

Usage: `from brian2.only import *`

Exported members: *`get_logger()`*, *`BrianLogger`*, *`std_silent`*, *`Trackable`*, *`Nameable`*, *`SpikeSource`*, *`linked_var()`*, `DEFAULT_FUNCTIONS`, *`Function`*, *`implementation()`*, *`declare_types()`*, *`PreferenceError`*, *`BrianPreference`*, *`prefs`*, *`brian_prefs`*, *`Clock`*, *`defaultclock`*, *`Equations`*, *`Expression`*, *`Statements`*, *`BrianObject`*, *`BrianObjectException`*, *`Network`*, *`profiling_summary()`*, *`scheduling_summary()`* … (304 more members)

Functions

restore_initial_state()	Restores internal Brian variables to the state they are in when Brian is imported

6.6.1 restore_initial_state function

(*Shortest import*: `from brian2 import restore_initial_state`)

`brian2.only.`**`restore_initial_state`**`()`
    Restores internal Brian variables to the state they are in when Brian is imported

    Resets `defaultclock.dt = 0.1*ms`, `BrianGlobalPreferences._restore` preferences, and set *`BrianObject._scope_current_key`* back to 0.

6.7 Subpackages

6.7.1 codegen package

Package providing the code generation framework.

Exported members: *NumpyCodeObject*, *CythonCodeObject*

`_prefs` module

Module declaring general code generation preferences.

Preferences

Code generation preferences

`codegen.loop_invariant_optimisations = True`

    Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/… Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.max_cache_dir_size = 1000`

    The size of a directory (in MB) with cached code for Cython that triggers a warning. Set to 0 to never get a warning.

`codegen.string_expression_target = 'numpy'`

    Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

    Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.

- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.

- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

codeobject **module**

Module providing the base *CodeObject* and related functions.

Exported members: *CodeObject*, *constant_or_scalar*

Classes

CodeObject(*args, **kw)	Executable code object.

CodeObject class

(*Shortest import*: `from brian2.codegen.codeobject import CodeObject`)

class brian2.codegen.codeobject.**CodeObject**(*args*, **kw*)

Bases: *brian2.core.names.Nameable*

Executable code object.

The `code` can either be a string or a *brian2.codegen.templates.MultiTemplate*.

After initialisation, the code is compiled with the given namespace using `code.compile(namespace)`.

Calling `code(key1=val1, key2=val2)` executes the code with the given variables inserted into the namespace.

Attributes

class_name	A short name for this type of *CodeObject*
generator_class	The `CodeGenerator` class used by this *CodeObject*

Methods

__call__(**kwds)	Call self as a function.
after_run()	Runs the finalizing code in the namespace.
before_run()	Runs the preparation code in the namespace.
compile()	

continues on next page

Table 19 – continued from previous page

compile_block(block)	
is_available()	Whether this target for code generation is available.
run()	Runs the main code in the namespace.
run_block(block)	
update_namespace()	Update the namespace for this timestep.

Details

class_name
> A short name for this type of *CodeObject*

generator_class
> The CodeGenerator class used by this *CodeObject*

__call__(***kwds*)
> Call self as a function.

after_run()
> Runs the finalizing code in the namespace. This code will only be executed once per run.
>
> > **Returns return_value** : dict
> >
> > > A dictionary with the keys corresponding to the output_variables defined during the call of CodeGenerator.code_object.

before_run()
> Runs the preparation code in the namespace. This code will only be executed once per run.
>
> > **Returns return_value** : dict
> >
> > > A dictionary with the keys corresponding to the output_variables defined during the call of CodeGenerator.code_object.

compile()

compile_block(*block*)

classmethod is_available()
> Whether this target for code generation is available. Should use a minimal example to check whether code generation works in general.

run()
> Runs the main code in the namespace.
>
> > **Returns return_value** : dict
> >
> > > A dictionary with the keys corresponding to the output_variables defined during the call of CodeGenerator.code_object.

run_block(*block*)

update_namespace()
> Update the namespace for this timestep. Should only deal with variables where *the reference* changes every timestep, i.e. where the current reference in namespace is not correct.

Functions

check_compiler_kwds(compiler_kwds, ...)	Internal function to check the provided compiler keywords against the list of understood keywords.

check_compiler_kwds function

(*Shortest import*: `from brian2.codegen.codeobject import check_compiler_kwds`)

brian2.codegen.codeobject.**check_compiler_kwds**(*compiler_kwds*, *accepted_kwds*, *target*)
    Internal function to check the provided compiler keywords against the list of understood keywords.

> **Parameters** **compiler_kwds** : dict
>
> > Dictionary of compiler keywords and respective list of values.
>
> **accepted_kwds** : list of str
>
> > The compiler keywords understood by the code generation target
>
> **target** : str
>
> > The name of the code generation target (used for the error message).

> ### Raises

> **ValueError** If a compiler keyword is not understood

constant_or_scalar(varname, variable)	Convenience function to generate code to access the value of a variable.

constant_or_scalar function

(*Shortest import*: `from brian2.codegen.codeobject import constant_or_scalar`)

brian2.codegen.codeobject.**constant_or_scalar**(*varname*, *variable*)
    Convenience function to generate code to access the value of a variable. Will return `'varname'` if the `variable` is a constant, and `array_name[0]` if it is a scalar array.

create_runner_codeobj(group, code, ...[, ...])	Create a *CodeObject* for the execution of code in the context of a *Group*.

create_runner_codeobj function

(*Shortest import*: `from brian2.codegen.codeobject import create_runner_codeobj`)

`brian2.codegen.codeobject.`**`create_runner_codeobj`**(*group*, *code*, *template_name*, *run_namespace*, *user_code=None*, *variable_indices=None*, *name=None*, *check_units=True*, *needed_variables=None*, *additional_variables=None*, *template_kwds=None*, *override_conditional_write=None*, *codeobj_class=None*)

Create a *CodeObject* for the execution of code in the context of a *Group*.

> **Parameters** **group** : *Group*
>
> > The group where the code is to be run
>
> **code** : str or dict of str
>
> > The code to be executed.
>
> **template_name** : str
>
> > The name of the template to use for the code.
>
> **run_namespace** : dict-like
>
> > An additional namespace that is used for variable lookup (either an explicitly defined namespace or one taken from the local context).
>
> **user_code** : str, optional
>
> > The code that had been specified by the user before other code was added automatically. If not specified, will be assumed to be identical to `code`.
>
> **variable_indices** : dict-like, optional
>
> > A mapping from `Variable` objects to index names (strings). If none is given, uses the corresponding attribute of `group`.
>
> **name** : str, optional
>
> > A name for this code object, will use `group` + `'_codeobject*'` if none is given.
>
> **check_units** : bool, optional
>
> > Whether to check units in the statement. Defaults to `True`.
>
> **needed_variables: list of str, optional** :
>
> > A list of variables that are neither present in the abstract code, nor in the `USES_VARIABLES` statement in the template. This is only rarely necessary, an example being a *StateMonitor* where the names of the variables are neither known to the template nor included in the abstract code statements.
>
> **additional_variables** : dict-like, optional
>
> > A mapping of names to `Variable` objects, used in addition to the variables saved in `group`.
>
> **template_kwds** : dict, optional
>
> > A dictionary of additional information that is passed to the template.
>
> **override_conditional_write: list of str, optional** :
>
> > A list of variable names which are used as conditions (e.g. for refractoriness) which should be ignored.

codeobj_class : class, optional

The [*CodeObject*](#) class to run code with. If not specified, defaults to the `group`'s `codeobj_class` attribute.

`cpp_prefs` module

Preferences related to C++ compilation

Preferences

C++ compilation preferences

`codegen.cpp.compiler = ''`

Compiler to use (uses default if empty). Should be `'unix'` or `'msvc'`.

To specify a specific compiler binary on unix systems, set the `CXX` environment variable instead.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args = None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvc`).

`codegen.cpp.extra_compile_args_gcc = ['-w', '-O3', '-ffast-math', '-fno-finite-math-only', '-march=native', '-std=c++11']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc = ['/Ox', '/w', '', '/MP']`

Extra compile arguments to pass to MSVC compiler (the default `/arch:` flag is determined based on the processor architecture)

`codegen.cpp.extra_link_args = []`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers = []`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header'"]. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs = ['/path/to/your/Python/environment/include']`

Include directories to use. The default value is `$prefix/include` (or `$prefix/Library/include` on Windows), where `$prefix` is Python's site-specific directory prefix as returned by [`sys.prefix`](#). This will make compilation use library files installed into a conda environment.

`codegen.cpp.libraries = []`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs = ['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at link time. The default value is `$prefix/lib` (or `$prefix/Library/lib` on Windows), where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

`codegen.cpp.msvc_architecture = ''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location = ''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs = ['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at run time. The default value is `$prefix/lib` (not used on Windows), where `$prefix` is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

Exported members: *`get_compiler_and_args`*, *`get_msvc_env`*, *`compiler_supports_c99`*, *`C99Check`*

Classes

`C99Check`(name)	Helper class to create objects that can be passed as an `availability_check` to a `FunctionImplementation`.

C99Check class

(*Shortest import*: `from brian2.codegen.cpp_prefs import C99Check`)

class `brian2.codegen.cpp_prefs.`**`C99Check`**(*name*)

Bases: `object`

Helper class to create objects that can be passed as an `availability_check` to a `FunctionImplementation`.

Methods

`__call__`(*args, **kwargs)	Call self as a function.

Details

`__call__`(**args*, ***kwargs*)
Call self as a function.

Functions

`compiler_supports_c99`()	

compiler_supports_c99 function

(*Shortest import*: `from brian2.codegen.cpp_prefs import compiler_supports_c99`)

brian2.codegen.cpp_prefs.**compiler_supports_c99**()

get_compiler_and_args()	Returns the computed compiler and compilation flags

get_compiler_and_args function

(*Shortest import*: `from brian2.codegen.cpp_prefs import get_compiler_and_args`)

brian2.codegen.cpp_prefs.**get_compiler_and_args**()
    Returns the computed compiler and compilation flags

get_msvc_env()	

get_msvc_env function

(*Shortest import*: `from brian2.codegen.cpp_prefs import get_msvc_env`)

brian2.codegen.cpp_prefs.**get_msvc_env**()

has_flag(compiler, flagname)	

has_flag function

(*Shortest import*: `from brian2.codegen.cpp_prefs import has_flag`)

brian2.codegen.cpp_prefs.**has_flag**(*compiler*, *flagname*)

`get_cpu_flags` module

This script is used to ask for the CPU flags on Windows. We use this instead of importing the cpuinfo package, because recent versions of py-cpuinfo use the multiprocessing module, and any import of cpuinfo that is not within a `if __name__ == '__main__':` block will lead to the script being executed twice.

The CPU flags are printed to stdout encoded as JSON.

`optimisation` module

Simplify and optimise sequences of statements by rewriting and pulling out loop invariants.

Exported members: `optimise_statements`, `ArithmeticSimplifier`, `Simplifier`

Classes

`ArithmeticSimplifier`(variables)	Carries out the following arithmetic simplifications:

ArithmeticSimplifier class

(*Shortest import*: `from brian2.codegen.optimisation import ArithmeticSimplifier`)

class `brian2.codegen.optimisation.`**`ArithmeticSimplifier`**(*variables*)

  Bases: `brian2.parsing.bast.BrianASTRenderer`

  Carries out the following arithmetic simplifications:

   1. Constant evaluation (e.g. exp(0)=1) by attempting to evaluate the expression in an "assumptions namespace"

   2. Binary operators, e.g. 0*x=0, 1*x=x, etc. You have to take care that the dtypes match here, e.g. if x is an integer, then 1.0*x shouldn't be replaced with x but left as 1.0*x.

    Parameters variables : dict of (str, Variable)

      Usual definition of variables.

     assumptions : sequence of str

      Additional assumptions that can be used in simplification, each assumption is a string statement. These might be the scalar statements for example.

  #### Methods

`render_BinOp`(node)	
`render_node`(node)	Assumes that the node has already been fully processed by BrianASTRenderer

  #### Details

  `render_BinOp`(*node*)

  `render_node`(*node*)
   Assumes that the node has already been fully processed by BrianASTRenderer

`Simplifier`(variables, scalar_statements[, ...])	Carry out arithmetic simplifications (see `ArithmeticSimplifier`) and loop invariants

Simplifier class

(*Shortest import*: `from brian2.codegen.optimisation import Simplifier`)

class brian2.codegen.optimisation.**Simplifier**(*variables*, *scalar_statements*, *extra_lio_prefix=''*)
Bases: *brian2.parsing.bast.BrianASTRenderer*

Carry out arithmetic simplifications (see *ArithmeticSimplifier*) and loop invariants

> **Parameters** **variables** : dict of (str, Variable)
>
>> Usual definition of variables.
>
>> **scalar_statements** : sequence of Statement
>
>> Predefined scalar statements that can be used as part of simplification

Notes

After calling *render_expr* on a sequence of expressions (coming from vector statements typically), this object will have some new attributes:

loop_invariants [OrderedDict of (expression, varname)] varname will be of the form `_lio_N` where `N` is some integer, and the expressions will be strings that correspond to scalar-only expressions that can be evaluated outside of the vector block.

loop_invariant_dtypes [dict of (varname, dtypename)] dtypename will be one of `'boolean'`, `'integer'`, `'float'`.

Methods

render_expr(expr)	
render_node(node)	Assumes that the node has already been fully processed by BrianASTRenderer

Details

render_expr(*expr*)

render_node(*node*)
Assumes that the node has already been fully processed by BrianASTRenderer

Functions

cancel_identical_terms(primary, inverted)	Cancel terms in a collection, e.g.

cancel_identical_terms function

(*Shortest import*: `from brian2.codegen.optimisation import cancel_identical_terms`)

`brian2.codegen.optimisation.`**`cancel_identical_terms`**(*primary*, *inverted*)
    Cancel terms in a collection, e.g. a+b-a should be cancelled to b

    Simply renders the nodes into expressions and removes whenever there is a common expression in primary and inverted.

> **Parameters primary** : list of AST nodes
>
>> These are the nodes that are positive with respect to the operator, e.g. in x*y/z it would be [x, y].
>
> **inverted** : list of AST nodes
>
>> These are the nodes that are inverted with respect to the operator, e.g. in x*y/z it would be [z].
>
> **Returns primary** : list of AST nodes
>
>> Primary nodes after cancellation
>
> **inverted** : list of AST nodes
>
>> Inverted nodes after cancellation

collect(node)	Attempts to collect commutative operations into one and simplifies them.

collect function

(*Shortest import*: `from brian2.codegen.optimisation import collect`)

`brian2.codegen.optimisation.`**`collect`**(*node*)
    Attempts to collect commutative operations into one and simplifies them.

    For example, if x and y are scalars, and z is a vector, then (x*z)*y should be rewritten as (x*y)*z to minimise the number of vector operations. Similarly, ((x*2)*3)*4 should be rewritten as x*24.

    Works for either multiplication/division or addition/subtraction nodes.

    The final output is a subexpression of the following maximal form:

> (((numerical_value*(product of scalars))/(product of scalars))*(product of vectors))/(product of vectors)

    Any possible cancellations will have been done.

> **Parameters node** : Brian AST node
>
>> The node to be collected/simplified.
>
> **Returns node** : Brian AST node
>
>> Simplified node.

collect_commutative(node, primary, inverted, ...)	

collect_commutative function

(*Shortest import*: `from brian2.codegen.optimisation import collect_commutative`)

`brian2.codegen.optimisation.`**`collect_commutative`**(*node*, *primary*, *inverted*, *terms_primary*, *terms_inverted*, *add_to_inverted=False*)

evaluate_expr(expr, ns)	Try to evaluate the expression in the given namespace

evaluate_expr function

(*Shortest import*: `from brian2.codegen.optimisation import evaluate_expr`)

`brian2.codegen.optimisation.`**`evaluate_expr`**(*expr*, *ns*)

Try to evaluate the expression in the given namespace

Returns either (value, True) if successful, or (expr, False) otherwise.

expression_complexity(expr, variables)	

expression_complexity function

(*Shortest import*: `from brian2.codegen.optimisation import expression_complexity`)

`brian2.codegen.optimisation.`**`expression_complexity`**(*expr*, *variables*)

optimise_statements(scalar_statements, ...)	Optimise a sequence of scalar and vector statements

optimise_statements function

(*Shortest import*: `from brian2.codegen.optimisation import optimise_statements`)

`brian2.codegen.optimisation.`**`optimise_statements`**(*scalar_statements*, *vector_statements*, *variables*, *blockname=''*)

Optimise a sequence of scalar and vector statements

Performs the following optimisations:

1. Constant evaluations (e.g. exp(0) to 1). See *evaluate_expr*.

2. Arithmetic simplifications (e.g. 0*x to 0). See *ArithmeticSimplifier*, *collect()*.

3. Pulling out loop invariants (e.g. v*exp(-dt/tau) to a=exp(-dt/tau) outside the loop and v*a inside). See *Simplifier*.

4. Boolean simplifications (allowing the replacement of expressions with booleans with a sequence of if/thens). See *Simplifier*.

> **Parameters** **scalar_statements** : sequence of Statement
>
> > Statements that only involve scalar values and should be evaluated in the scalar block.
>
> **vector_statements** : sequence of Statement

Statements that involve vector values and should be evaluated in the vector block.

> **variables** : dict of (str, Variable)
>
> > Definition of the types of the variables.
>
> **blockname** : str, optional
>
> > Name of the block (used for LIO constant prefixes to avoid name clashes)
>
> **Returns new_scalar_statements** : sequence of Statement
>
> > As above but with loop invariants pulled out from vector statements
>
> **new_vector_statements** : sequence of Statement
>
> > Simplified/optimised versions of statements

reduced_node(terms, op)	Reduce a sequence of terms with the given operator

reduced_node function

(*Shortest import*: `from brian2.codegen.optimisation import reduced_node`)

`brian2.codegen.optimisation.`**`reduced_node`**(*terms*, *op*)

> Reduce a sequence of terms with the given operator
>
> For examples, if terms were [a, b, c] and op was multiplication then the reduction would be (a*b)*c.
>
> > **Parameters terms** : list
> >
> > > AST nodes.
> >
> > **op** : AST node
> >
> > > Could be `ast.Mult` or `ast.Add`.

Examples

```
>>> import ast
>>> nodes = [ast.Name(id='x'), ast.Name(id='y'), ast.Name(id='z')]
>>> ast.dump(reduced_node(nodes, ast.Mult), annotate_fields=False)
"BinOp(BinOp(Name('x'), Mult(), Name('y')), Mult(), Name('z'))"
>>> nodes = [ast.Name(id='x')]
>>> ast.dump(reduced_node(nodes, ast.Add), annotate_fields=False)
"Name('x')"
```

permutation_analysis module

Module for analysing synaptic pre and post code for synapse order independence.

Exported members: *OrderDependenceError*, *check_for_order_independence*

Classes

OrderDependenceError	

OrderDependenceError class

(*Shortest import*: `from brian2.codegen.permutation_analysis import OrderDependenceError`)

class `brian2.codegen.permutation_analysis.`**`OrderDependenceError`**
    Bases: `Exception`

Functions

check_for_order_independence(statements, ...)	Check that the sequence of statements doesn't depend on the order in which the indices are iterated through.

check_for_order_independence function

(*Shortest import*: `from brian2.codegen.permutation_analysis import check_for_order_independence`)

`brian2.codegen.permutation_analysis.`**`check_for_order_independence`**(*statements*, *variables*, *indices*)
    Check that the sequence of statements doesn't depend on the order in which the indices are iterated through.

statements module

Module providing the *Statement* class.

Classes

Statement(var, op, expr, comment, dtype[, ...])	A single line mathematical statement.

Statement class

(*Shortest import*: `from brian2.codegen.statements import Statement`)

class `brian2.codegen.statements.`**`Statement`**(*var*, *op*, *expr*, *comment*, *dtype*, *constant=False*, *subexpression=False*, *scalar=False*)

    Bases: `object`

    A single line mathematical statement.

    The structure is `var op expr`.

    Parameters **var** : str

            The left hand side of the statement, the value being written to.

    op : str

            The operation, can be any of the standard Python operators (including += etc.) or a special operator `:=` which means you are defining a new symbol (whereas = means you are setting

the value of an existing symbol).

expr : str, *Expression*

The right hand side of the statement.

dtype : `dtype`

The numpy dtype of the value or array `var()`.

constant : bool, optional

Set this flag to `True` if the value will not change (only applies for `op=='':='`.

subexpression : bool, optional

Set this flag to `True` if the variable is a subexpression. In some languages (e.g. Python) you can use this to save a memory copy, because you don't need to do `lhs[:]  = rhs` but a redefinition `lhs = rhs`.

scalar : bool, optional

Set this flag to `True` if `var()` and `expr` are scalar.

Notes

Will compute the following attribute:

inplace True or False depending if the operation is in-place or not.

Boolean simplification notes:

Will initially set the attribute `used_boolean_variables` to `None`. This is set by *opti-mise_statements* when it is called on a sequence of statements to the list of boolean variables that are used in this expression. In addition, the attribute `boolean_simplified_expressions` is set to a dictionary with keys consisting of a tuple of pairs `(var,  value)` where `var` is the name of the boolean variable (will be in `used_boolean_variables`) and `var` is `True` or `False`. The values of the dictionary are strings representing the simplified version of the expression if each `var=value` substitution is made for that key. The keys will range over all possible values of the set of boolean variables. The complexity of the original statement is set as the attribute `complexity_std`, and the complexity of the simplified versions are in the dictionary `complexities` (with the same keys).

This information can be used to generate code that replaces a complex expression that varies depending on the value of one or more boolean variables with an `if/then` sequence where each subexpression is simplified. It is optional to use this (e.g. the numpy codegen does not, but the cython one does).

`targets` **module**

Module that stores all known code generation targets as `codegen_targets`.

Exported members: `codegen_targets`

templates module

Handles loading templates from a directory.

Exported members: *Templater*

Classes

CodeObjectTemplate(template, template_source)	Single template object returned by *Templater* and used for final code generation

CodeObjectTemplate class

(*Shortest import*: `from brian2.codegen.templates import CodeObjectTemplate`)

class `brian2.codegen.templates.`**`CodeObjectTemplate`**(*template*, *template_source*)
   Bases: *object*

   Single template object returned by *Templater* and used for final code generation

   Should not be instantiated by the user, but only directly by *Templater*.

   #### **Notes**

   The final code is obtained from this by calling the template (see __call__).

   #### **Attributes**

allows_scalar_write	Does this template allow writing to scalar variables?
iterate_all	The indices over which the template iterates completely
variables	The set of variables in this template
writes_read_only	Read-only variables that are changed by this template

   #### **Methods**

__call__(scalar_code, vector_code, **kwds)	Return a usable code block or blocks from this template.

   #### **Details**

   allows_scalar_write
      Does this template allow writing to scalar variables?

   iterate_all
      The indices over which the template iterates completely

   variables
      The set of variables in this template

writes_read_only
    Read-only variables that are changed by this template

__call__(*scalar_code*, *vector_code*, ***kwds*)
    Return a usable code block or blocks from this template.

> **Parameters** **scalar_code** : dict
>
> > Dictionary of scalar code blocks.
>
> **vector_code** : dict
>
> > Dictionary of vector code blocks
>
> ****kwds** :
>
> > Additional parameters to pass to the template

Notes

Returns either a string (if macros were not used in the template), or a *MultiTemplate* (if macros were used).

LazyTemplateLoader(environment, extension)	Helper object to load templates only when they are needed.

LazyTemplateLoader class

(*Shortest import*: `from brian2.codegen.templates import LazyTemplateLoader`)

class brian2.codegen.templates.**LazyTemplateLoader**(*environment*, *extension*)
    Bases: `object`

Helper object to load templates only when they are needed.

Methods

get_template(name)	

Details

get_template(*name*)

MultiTemplate(module)	Code generated by a *CodeObjectTemplate* with multiple blocks

---

MultiTemplate class

(*Shortest import*: `from brian2.codegen.templates import MultiTemplate`)

class `brian2.codegen.templates.`**`MultiTemplate`**(*module*)

Bases: `collections.abc.Mapping`

Code generated by a *`CodeObjectTemplate`* with multiple blocks

Each block is a string stored as an attribute with the block name. The object can also be accessed as a dictionary.

Templater(package_name, extension[, ...])	Class to load and return all the templates a `CodeObject` defines.

Templater class

(*Shortest import*: `from brian2.codegen.templates import Templater`)

class `brian2.codegen.templates.`**`Templater`**(*package_name*, *extension*, *env_globals=None*, *templates_dir='templates'*)

Bases: `object`

Class to load and return all the templates a `CodeObject` defines.

> **Parameters** **package_name** : str, tuple of str
>
>> The package where the templates are saved. If this is a tuple then each template will be searched in order starting from the first package in the tuple until the template is found. This allows for derived templates to be used. See also *`derive`*.
>
> **extension** : str
>
>> The file extension (e.g. `.pyx`) used for the templates.
>
> **env_globals** : dict (optional)
>
>> A dictionary of global values accessible by the templates. Can be used for providing utility functions. In all cases, the filter 'autoindent' is available (see existing templates for example usage).
>
> **templates_dir** : str, tuple of str, optional
>
>> The name of the directory containing the templates. Defaults to `'templates'`.

Notes

Templates are accessed using `templater.template_base_name` (the base name is without the file extension). This returns a *`CodeObjectTemplate`*.

Methods

[*derive*](package_name[, extension, ...])	Return a new Templater derived from this one, where the new package name and globals overwrite the old.

Details

derive (*package_name*, *extension=None*, *env_globals=None*, *templates_dir='templates'*)
> Return a new Templater derived from this one, where the new package name and globals overwrite the old.

Functions

[*autoindent*](code)

autoindent function

(*Shortest import*: `from brian2.codegen.templates import autoindent`)

`brian2.codegen.templates.`**`autoindent`**(*code*)

[*autoindent_postfilter*](code)

autoindent_postfilter function

(*Shortest import*: `from brian2.codegen.templates import autoindent_postfilter`)

`brian2.codegen.templates.`**`autoindent_postfilter`**(*code*)

[*variables_to_array_names*](variables[, ...])

variables_to_array_names function

(*Shortest import*: `from brian2.codegen.templates import variables_to_array_names`)

`brian2.codegen.templates.`**`variables_to_array_names`**(*variables*, *access_data=True*)

translation module

This module translates a series of statements into a language-specific syntactically correct code block that can be inserted into a template.

It infers whether or not a variable can be declared as constant, etc. It should handle common subexpressions, and so forth.

The input information needed:

- The sequence of statements (a multiline string) in standard mathematical form

- The list of known variables, common subexpressions and functions, and for each variable whether or not it is a value or an array, and if an array what the dtype is.

- The dtype to use for newly created variables

- The language to translate to

Exported members: *analyse_identifiers*, *get_identifiers_recursively*

Classes

LineInfo(**kwds)	A helper class, just used to store attributes.

LineInfo class

(*Shortest import*: `from brian2.codegen.translation import LineInfo`)

class `brian2.codegen.translation.`**`LineInfo`**(***kwds*)
    Bases: `object`

    A helper class, just used to store attributes.

Functions

analyse_identifiers(code, variables[, recursive])	Analyses a code string (sequence of statements) to find all identifiers by type.

analyse_identifiers function

(*Shortest import*: `from brian2.codegen.translation import analyse_identifiers`)

`brian2.codegen.translation.`**`analyse_identifiers`**(*code*, *variables*, *recursive=False*)
    Analyses a code string (sequence of statements) to find all identifiers by type.

    In a given code block, some variable names (identifiers) must be given as inputs to the code block, and some are created by the code block. For example, the line:

```
a = b+c
```

    This could mean to create a new variable a from b and c, or it could mean modify the existing value of a from b or c, depending on whether a was previously known.

        Parameters code : str

            The code string, a sequence of statements one per line.

        variables : dict of `Variable`, set of names

Specifiers for the model variables or a set of known names

recursive : bool, optional

Whether to recurse down into subexpressions (defaults to `False`).

Returns newly_defined : set

A set of variables that are created by the code block.

used_known : set

A set of variables that are used and already known, a subset of the `known` parameter.

unknown : set

A set of variables which are used by the code block but not defined by it and not previously known. Should correspond to variables in the external namespace.

get_identifiers_recursively(expressions, ...)	Gets all the identifiers in a list of expressions, recursing down into subexpressions.

get_identifiers_recursively function

(*Shortest import*: `from brian2.codegen.translation import get_identifiers_recursively`)

`brian2.codegen.translation.`**`get_identifiers_recursively`**(*expressions*, *variables*, *include_numbers=False*)

Gets all the identifiers in a list of expressions, recursing down into subexpressions.

Parameters expressions : list of str

List of expressions to check.

variables : dict-like

Dictionary of `Variable` objects

include_numbers : bool, optional

Whether to include number literals in the output. Defaults to `False`.

is_scalar_expression(expr, variables)	Whether the given expression is scalar.

is_scalar_expression function

(*Shortest import*: `from brian2.codegen.translation import is_scalar_expression`)

`brian2.codegen.translation.`**`is_scalar_expression`**(*expr*, *variables*)

Whether the given expression is scalar.

Parameters expr : str

The expression to check

variables : dict-like

`Variable` and *Function* object for all the identifiers used in `expr`

Returns scalar : bool

Whether `expr` is a scalar expression

[make_statements](code, variables, dtype[, ...])	Turn a series of abstract code statements into Statement objects, inferring whether each line is a set/declare operation, whether the variables are constant or not, and handling the cacheing of subexpressions.

make_statements function

(*Shortest import*: `from brian2.codegen.translation import make_statements`)

`brian2.codegen.translation.`**`make_statements`**(*code*, *variables*, *dtype*, *optimise=True*, *blockname=''*)

Turn a series of abstract code statements into Statement objects, inferring whether each line is a set/declare operation, whether the variables are constant or not, and handling the cacheing of subexpressions.

Parameters code : str

A (multi-line) string of statements.

variables : dict-like

A dictionary of with `Variable` and `Function` objects for every identifier used in the `code`.

dtype : `dtype`

The data type to use for temporary variables

optimise : bool, optional

Whether to optimise expressions, including pulling out loop invariant expressions and putting them in new scalar constants. Defaults to `False`, since this function is also used just to in contexts where we are not interested by this kind of optimisation. For the main code generation stage, its value is set by the *codegen.loop_invariant_optimisations* preference.

blockname : str, optional

A name for the block (used to name intermediate variables to avoid name clashes when multiple blocks are used together)

Returns :

——- :

scalar_statements, vector_statements : (list of `Statement`, list of `Statement`)

Lists with statements that are to be executed once and statements that are to be executed once for every neuron/synapse/… (or in a vectorised way)

Notes

If `optimise` is `True`, then the `scalar_statements` may include newly introduced scalar constants that have been identified as loop-invariant and have therefore been pulled out of the vector statements. The resulting statements will also use augmented assignments where possible, i.e. a statement such as `w = w + 1` will be replaced by `w += 1`. Also, statements involving booleans will have additional information added to them (see `Statement` for details) describing how the statement can be reformulated as a sequence of if/then statements. Calls *optimise_statements*.

Subpackages

generators package

`GSL_generator` module

GSLCodeGenerators for code that uses the ODE solver provided by the GNU Scientific Library (GSL)

Exported members: *GSLCodeGenerator*, *GSLCPPCodeGenerator*, *GSLCythonCodeGenerator*

Classes

GSLCPPCodeGenerator(variables, ...[, ...])	

<div align="center">

Methods

</div>

GSLCPPCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import GSLCPPCodeGenerator`)

class brian2.codegen.generators.GSL_generator.**GSLCPPCodeGenerator**(*variables, variable_indices, owner, iterate_all, codeobj_class, name, template_name, override_conditional_write=None, allows_scalar_write=False*)

    Bases: *brian2.codegen.generators.GSL_generator.GSLCodeGenerator*

Methods

c_data_type(dtype)	Get string version of object dtype that is attached to Brian variables.
initialize_array(varname, values)	Initialize a static array with given floating point values.
unpack_namespace_single(var_obj, in_vector, ...)	Writes the code necessary to pull single variable out of the Brian namespace into the generated code.
var_init_lhs(var, type)	Get string version of the left hand side of an initializing expression
var_replace_diff_var_lhs(var, ind)	

Details

c_data_type(*dtype*)

Get string version of object dtype that is attached to Brian variables. c pp_generator already has this function, but the Cython generator does not, but we need it for GSL code generation.

initialize_array(*varname*, *values*)

Initialize a static array with given floating point values. E.g. in C++, when called with arguments `array` and `[1.0, 3.0, 2.0]`, this method should return `double array[] = {1.0, 3.0, 2.0}`.

> **Parameters varname** : str
>
> > The name of the array variable that should be initialized
>
> **values** : list of float
>
> > The values that should be assigned to the array
>
> **Returns code** : str
>
> > One or more lines of array initialization code.

unpack_namespace_single(*var_obj*, *in_vector*, *in_scalar*)

Writes the code necessary to pull single variable out of the Brian namespace into the generated code.

The code created is significantly different between cpp and cython, so I decided to not make this function general over all target languages (i.e. in contrast to most other functions that only have syntactical differences)

var_init_lhs(*var*, *type*)

Get string version of the left hand side of an initializing expression

> **Parameters var** : str
>
> > **type** : str
>
> **Returns code** : str
>
> > For cpp returns type + var, while for cython just var

var_replace_diff_var_lhs(*var*, *ind*)

[*GSLCodeGenerator*](variables, ...[, ...])   GSL code generator.

GSLCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import GSLCodeGenerator`)

class brian2.codegen.generators.GSL_generator.**GSLCodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: [`object`](#)

GSL code generator.

Notes

Approach is to first let the already existing code generator for a target language do the bulk of the translating from abstract_code to actual code. This generated code is slightly adapted to render it GSL compatible. The most critical part here is that the vector_code that is normally contained in a loop in the `main()` is moved to the function `_GSL_func` that is sent to the GSL integrator. The variables used in the vector_code are added to a struct named `dataholder` and their values are set from the Brian namespace just before the scalar code block.

Methods

add_gsl_variables_as_non_scalar(diff_vars)	Add _gsl variables as non-scalar.
add_meta_variables(options)	
c_data_type(dtype)	Get string version of object dtype that is attached to Brian variables.
diff_var_to_replace(diff_vars)	Add differential variable-related strings that need to be replaced to go from normal brian to GSL code
find_differential_variables(code)	Find the variables that were tagged _gsl_{var}_f{ind} and return var, ind pairs.
find_function_names()	Return a list of used function names in the self.variables dictionary
find_undefined_variables(statements)	Find identifiers that are not in `self.variables` dictionary.
find_used_variables(statements, other_variables)	Find all the variables used in the right hand side of the given expressions.
get_dimension_code(diff_num)	Generate code for function that sets the dimension of the ODE system.
initialize_array(varname, values)	Initialize a static array with given floating point values.
is_constant_and_cpp_standalone(var_obj)	Check whether self.cpp_standalone and variable is Constant.
is_cpp_standalone()	Check whether we're running with cpp_standalone.
make_function_code(lines)	Add lines of GSL translated vector code to 'non-changing' _GSL_func code.
scale_array_code(diff_vars, method_options)	Return code for definition of _GSL_scale_array in generated code.
to_replace_vector_vars(variables_in_vector)	Create dictionary containing key, value pairs with to be replaced text to translate from conventional Brian to GSL.
translate(code, dtype)	Translates an abstract code block into the target language.
translate_scalar_code(code_lines, ...)	Translate scalar code: if calculated variables are used in the vector_code their value is added to the variable in the _dataholder.
translate_vector_code(code_lines, to_replace)	Translate vector code to GSL compatible code by substituting fragments of code.
unpack_namespace(variables_in_vector, ...[, ...])	Write code that unpacks Brian namespace to cython/cpp namespace.
unpack_namespace_single(var_obj, in_vector, ...)	Writes the code necessary to pull single variable out of the Brian namespace into the generated code.
var_init_lhs(var, type)	Get string version of the left hand side of an initializing expression

continues on next page

Table 62 – continued from previous page

write_dataholder(variables_in_vector)	Return string with full code for _dataholder struct.
write_dataholder_single(var_obj)	Return string declaring a single variable in the `_dataholder` struct.
yvector_code(diff_vars)	Generate code for function dealing with GSLs y vector.

Details

add_gsl_variables_as_non_scalar(*diff_vars*)

Add _gsl variables as non-scalar.

In `GSLStateUpdater` the differential equation variables are substituted with GSL tags that describe the information needed to translate the conventional Brian code to GSL compatible code. This function tells Brian that the variables that contain these tags should always be vector variables. If we don't do this, Brian renders the tag-variables as scalar if no vector variables are used in the right hand side of the expression.

> **Parameters diff_vars** : dict
>
> > dictionary with variables as keys and differential equation index as value

add_meta_variables(*options*)

c_data_type(*dtype*)

Get string version of object dtype that is attached to Brian variables. c pp_generator already has this function, but the Cython generator does not, but we need it for GSL code generation.

diff_var_to_replace(*diff_vars*)

Add differential variable-related strings that need to be replaced to go from normal brian to GSL code

From the code generated by Brian's 'normal' generators (cpp_generator or cython_generator a few bits of text need to be replaced to get GSL compatible code. The bits of text related to differential equation variables are put in the replacer dictionary in this function.

> **Parameters diff_vars** : dict
>
> > dictionary with variables as keys and differential equation index as value
>
> **Returns to_replace** : dict
>
> > dictionary with strings that need to be replaced as keys and the strings that will replace them as values

find_differential_variables(*code*)

Find the variables that were tagged _gsl_{var}_f{ind} and return var, ind pairs.

`GSLStateUpdater` tagged differential variables and here we extract the information given in these tags.

> **Parameters code** : list of strings
>
> > A list of strings containing gsl tagged variables
>
> **Returns diff_vars** : dict
>
> > A dictionary with variable names as keys and differential equation index as value

find_function_names()

Return a list of used function names in the self.variables dictionary

Functions need to be ignored in the GSL translation process, because the brian generator already sufficiently dealt with them. However, the brian generator also removes them from the variables dict, so there is no way

to check whether an identifier is a function after the brian translation process. This function is called before this translation process and the list of function names is stored to be used in the GSL translation.

> **Returns function_names** : list

>> list of strings that are function names used in the code

find_undefined_variables(*statements*)
    Find identifiers that are not in `self.variables` dictionary.

Brian does not save the _lio_ variables it uses anywhere. This is problematic for our GSL implementation because we save the lio variables in the `_dataholder` struct (for which we need the datatype of the variables). This function adds the left hand side variables that are used in the vector code to the variable dictionary as `AuxiliaryVariables` (all we need later is the datatype).

> **Parameters statements** : list

>> list of statement objects (need to have the dtype attribute)

Notes

I keep `self.variables` and `other_variables` separate so I can distinguish what variables are in the Brian namespace and which ones are defined in the code itself.

find_used_variables(*statements*, *other_variables*)
    Find all the variables used in the right hand side of the given expressions.

> **Parameters statements** : list

>> list of statement objects

> **Returns used_variables** : dict

>> dictionary of variables that are used as variable name (str), `Variable` pairs.

get_dimension_code(*diff_num*)
    Generate code for function that sets the dimension of the ODE system.

GSL needs to know how many differential variables there are in the ODE system. Since the current approach is to have the code in the vector loop the same for all simulations, this dimension is set by an external function. The code for this set_dimension functon is written here. It is assumed the code will be the same for each target language with the exception of some syntactical differences

> **Parameters diff_num** : int

>> Number of differential variables that describe the ODE system

> **Returns set_dimension_code** : str

>> The code describing the target language function in a single string

initialize_array(*varname*, *values*)
    Initialize a static array with given floating point values. E.g. in C++, when called with arguments `array` and `[1.0, 3.0, 2.0]`, this method should return `double array[] = {1.0, 3.0, 2.0}`.

> **Parameters varname** : str

>> The name of the array variable that should be initialized

> **values** : list of float

>> The values that should be assigned to the array

> **Returns code** : str

One or more lines of array initialization code.

is_constant_and_cpp_standalone(*var_obj*)
> Check whether self.cpp_standalone and variable is Constant.
>
> This check is needed because in the case of using the cpp_standalone device we do not want to apply our GSL variable conversion (var –> _GSL_dataholder.var), because the cpp_standalone code generation process involves replacing constants with their actual value ('freezing'). This results in code that looks like (if for example var = 1.2): _GSL_dataholder.1.2 = 1.2 and _GSL_dataholder->1.2. To prevent repetitive calls to get_device() etc. the outcome of is_cpp_standalone is saved.
>
> > **Parameters var_obj** : `Variable`
> >
> > > instance of brian Variable class describing the variable
> >
> > **Returns is_cpp_standalone** : bool
> >
> > > whether the used device is cpp_standalone and the given variable is an instance of Constant

is_cpp_standalone()
> Check whether we're running with cpp_standalone.
>
> Test if `get_device()` is instance `CPPStandaloneDevice`.
>
> > **Returns is_cpp_standalone** : bool
> >
> > > whether currently using cpp_standalone device
>
> **See also:**
>
> [*is_constant_and_cpp_standalone*](#) uses the returned value

make_function_code(*lines*)
> Add lines of GSL translated vector code to 'non-changing' _GSL_func code.
>
> Adds nonchanging aspects of GSL _GSL_func code to lines of code written somewhere else ([*translate_vector_code*](#)). Here these lines are put between the non-changing parts of the code and the target-language specific syntax is added.
>
> > **Parameters lines** : str
> >
> > > Code containing GSL version of equations
> >
> > **Returns function_code** : str
> >
> > > code describing _GSL_func that is sent to GSL integrator.

scale_array_code(*diff_vars*, *method_options*)
> Return code for definition of _GSL_scale_array in generated code.
>
> > **Parameters diff_vars** : dict
> >
> > > dictionary with variable name (str) as key and differential variable index (int) as value
> >
> > > **method_options** : dict
> > >
> > > > dictionary containing integrator settings
> >
> > **Returns code** : str
> >
> > > full code describing a function returning a array containing doubles with the absolute errors for each differential variable (according to their assigned index in the GSL StateUpdater)

to_replace_vector_vars(*variables_in_vector*, *ignore=frozenset({})*)
> Create dictionary containing key, value pairs with to be replaced text to translate from conventional Brian to GSL.

> **Parameters variables_in_vector** : dict
>
>> dictionary with variable name (str), `Variable` pairs of variables occurring in vector code
>
>> **ignore** : set, optional
>>
>> set of strings with variable names that should be ignored
>
> **Returns to_replace** : dict
>
>> dictionary with strings that need to be replaced i.e. _lio_1 will be _GSL_dataholder._lio_1 (in cython) or _GSL_dataholder->_lio_1 (cpp)

Notes

t will always be added because GSL defines its own t. i.e. for cpp: {'const t = _ptr_array_defaultclock_t[0];' : ''}

translate (*code*, *dtype*)

Translates an abstract code block into the target language.

translate_scalar_code (*code_lines*, *variables_in_scalar*, *variables_in_vector*)

Translate scalar code: if calculated variables are used in the vector_code their value is added to the variable in the _dataholder.

> **Parameters code_lines** : list
>
>> list of strings containing scalar code
>
>> **variables_in_vector** : dict
>>
>> dictionary with variable name (str), `Variable` pairs of variables occurring in vector code
>
>> **variables_in_scalar** : dict
>>
>> dictionary with variable name (str), `Variable` pairs of variables occurring in scalar code
>
> **Returns scalar_code** : str
>
>> code fragment that should be injected in the main before the loop

translate_vector_code (*code_lines*, *to_replace*)

Translate vector code to GSL compatible code by substituting fragments of code.

> **Parameters code_lines** : list
>
>> list of strings describing the vector_code
>
>> **to_replace: dict** :
>>
>> dictionary with to be replaced strings (see to_replace_vector_vars and to_replace_diff_vars)
>
> **Returns vector_code** : str
>
>> New code that is now to be added to the function that is sent to the GSL integrator

unpack_namespace (*variables_in_vector*, *variables_in_scalar*, *ignore=frozenset({})*)

Write code that unpacks Brian namespace to cython/cpp namespace.

For vector code this means putting variables in _dataholder (i.e. _GSL_dataholder->var or _GSL_dataholder.var = ...) Note that code is written so a variable could occur both in scalar and vector code

> **Parameters variables_in_vector** : dict

dictionary with variable name (str), `Variable` pairs of variables occurring in vector code

variables_in_scalar : dict

dictionary with variable name (str), `Variable` pairs of variables occurring in scalar code

ignore : set, optional

set of string names of variables that should be ignored

Returns unpack_namespace_code : str

code fragment unpacking the Brian namespace (setting variables in the _dataholder struct in case of vector)

unpack_namespace_single(*var_obj*, *in_vector*, *in_scalar*)
Writes the code necessary to pull single variable out of the Brian namespace into the generated code.

The code created is significantly different between cpp and cython, so I decided to not make this function general over all target languages (i.e. in contrast to most other functions that only have syntactical differences)

var_init_lhs(*var*, *type*)
Get string version of the left hand side of an initializing expression

Parameters var : str

type : str

Returns code : str

For cpp returns type + var, while for cython just var

write_dataholder(*variables_in_vector*)
Return string with full code for _dataholder struct.

Parameters variables_in_vector : dict

dictionary containing variable name as key and `Variable` as value

Returns code : str

code for _dataholder struct

write_dataholder_single(*var_obj*)
Return string declaring a single variable in the `_dataholder` struct.

Parameters var_obj : `Variable`

Returns code : str

string describing this variable object as required for the `_dataholder` struct (e.g. `double* _array_neurongroup_v`)

yvector_code(*diff_vars*)
Generate code for function dealing with GSLs y vector.

The values of differential variables have to be transferred from Brian's namespace to a vector that is given to GSL. The transferring from Brian –> y and back from y –> Brian after integration happens in separate functions. The code for these is written here.

Parameters diff_vars : dictionary

Dictionary containing variable names as keys (str) and differential variable index as value

Returns yvector_code : str

The code for the two functions (`_fill_y_vector` and `_empty_y_vector`) as single string.

GSLCythonCodeGenerator(variables, ...[, ...])

GSLCythonCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import GSLCythonCodeGenerator`)

class brian2.codegen.generators.GSL_generator.**GSLCythonCodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: *brian2.codegen.generators.GSL_generator.GSLCodeGenerator*

Methods

c_data_type(dtype)	Get string version of object dtype that is attached to Brian variables.
get_array_name(var[, access_data])	
initialize_array(varname, values)	Initialize a static array with given floating point values.
unpack_namespace_single(var_obj, in_vector, ...)	Writes the code necessary to pull single variable out of the Brian namespace into the generated code.
var_init_lhs(var, type)	Get string version of the left hand side of an initializing expression
var_replace_diff_var_lhs(var, ind)	

Details

c_data_type(*dtype*)
    Get string version of object dtype that is attached to Brian variables. c pp_generator already has this function, but the Cython generator does not, but we need it for GSL code generation.

static get_array_name(*var*, *access_data=True*)

initialize_array(*varname*, *values*)
    Initialize a static array with given floating point values. E.g. in C++, when called with arguments `array` and `[1.0, 3.0, 2.0]`, this method should return `double array[] = {1.0, 3.0, 2.0}`.

> > **Parameters** **varname** : str
> >
> > > The name of the array variable that should be initialized
> >
> > **values** : list of float
> >
> > > The values that should be assigned to the array
> >
> > **Returns** **code** : str
> >
> > > One or more lines of array initialization code.

> **unpack_namespace_single**(*var_obj*, *in_vector*, *in_scalar*)
> Writes the code necessary to pull single variable out of the Brian namespace into the generated code.
>
> The code created is significantly different between cpp and cython, so I decided to not make this function general over all target languages (i.e. in contrast to most other functions that only have syntactical differences)

> **var_init_lhs**(*var*, *type*)
> Get string version of the left hand side of an initializing expression
>
> > **Parameters** **var** : str
> >
> > **type** : str
> >
> > **Returns** **code** : str
> >
> > > For cpp returns type + var, while for cython just var

> **var_replace_diff_var_lhs**(*var*, *ind*)

Functions

valid_gsl_dir(val)	Validate given string to be path containing required GSL files.

valid_gsl_dir function

(*Shortest import*: `from brian2.codegen.generators.GSL_generator import valid_gsl_dir`)

`brian2.codegen.generators.GSL_generator.`**valid_gsl_dir**(*val*)
Validate given string to be path containing required GSL files.

base module

Base class for generating code in different programming languages, gives the methods which should be overridden to implement a new language.

Exported members: *CodeGenerator*

Classes

CodeGenerator(variables, variable_indices, ...)	Base class for all languages.

CodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import CodeGenerator`)

class brian2.codegen.generators.base.**CodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

> Bases: `object`
>
> Base class for all languages.
>
> See definition of methods below.
>
> TODO: more details here

Methods

[`array_read_write`](statements)	Helper function, gives the set of ArrayVariables that are read from and written to in the series of statements.
[`arrays_helper`](statements)	Combines the two helper functions [`array_read_write`](#) and [`get_conditional_write_vars`](#), and updates the `read` set.
[`determine_keywords`]()	A dictionary of values that is made available to the templated.
[`get_array_name`](var[, access_data])	Get a globally unique name for a `ArrayVariable`.
[`get_conditional_write_vars`]()	Helper function, returns a dict of mappings (`varname`, `condition_var_name`) indicating that when `varname` is written to, it should only be when `condition_var_name` is `True`.
[`has_repeated_indices`](statements)	Whether any of the statements potentially uses repeated indices (e.g.
[`translate`](code, dtype)	Translates an abstract code block into the target language.
[`translate_expression`](expr)	Translate the given expression string into a string in the target language, returns a string.
[`translate_one_statement_sequence`](statements)	
[`translate_statement`](statement)	Translate a single line `Statement` into the target language, returns a string.
[`translate_statement_sequence`](...)	Translate a sequence of `Statement` into the target language, taking care to declare variables, etc.

Details

array_read_write(*statements*)

Helper function, gives the set of ArrayVariables that are read from and written to in the series of statements. Returns the pair read, write of sets of variable names.

arrays_helper(*statements*)

Combines the two helper functions *array_read_write* and *get_conditional_write_vars*, and updates the `read` set.

determine_keywords()

A dictionary of values that is made available to the templated. This is used for example by the CPPCode-Generator to set up all the supporting code

static get_array_name(*var*, *access_data=True*)

Get a globally unique name for a ArrayVariable.

> **Parameters var** : ArrayVariable
>
>> The variable for which a name should be found.
>
>> **access_data** : bool, optional
>
>> For DynamicArrayVariable objects, specifying True here means the name for the underlying data is returned. If specifying False, the name of object itself is returned (e.g. to allow resizing).
>
>> **Returns** :
>
>> **———-** :
>
>> **name** : str
>
>> A uniqe name for var().

get_conditional_write_vars()

Helper function, returns a dict of mappings (varname, condition_var_name) indicating that when varname is written to, it should only be when condition_var_name is True.

has_repeated_indices(*statements*)

Whether any of the statements potentially uses repeated indices (e.g. pre- or postsynaptic indices).

translate(*code*, *dtype*)

Translates an abstract code block into the target language.

translate_expression(*expr*)

Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)

Translate a single line Statement into the target language, returns a string.

translate_statement_sequence(*scalar_statements*, *vector_statements*)

Translate a sequence of Statement into the target language, taking care to declare variables, etc. if necessary.

Returns a tuple (scalar_code, vector_code, kwds) where scalar_code is a list of the lines of code executed before the inner loop, vector_code is a list of the lines of code in the inner loop, and kwds is a dictionary of values that is made available to the template.

`cpp_generator` module

Exported members: *[CPPCodeGenerator](#)*, *[c_data_type](#)*

Classes

[CPPCodeGenerator](#)(*args, **kwds)	C++ language

CPPCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import CPPCodeGenerator`)

class `brian2.codegen.generators.cpp_generator.`**CPPCodeGenerator**(**args*, ***kwds*)

 Bases: *[brian2.codegen.generators.base.CodeGenerator](#)*

 C++ language

 C++ code templates should provide Jinja2 macros with the following names:

 main The main loop.

 support_code The support code (function definitions, etc.), compiled in a separate file.

 For user-defined functions, there are two keys to provide:

 support_code The function definition which will be added to the support code.

 hashdefine_code The `#define` code added to the main loop.

 See *[TimedArray](#)* for an example of these keys.

Attributes

[flush_denormals](#)
[restrict](#)

Methods

[denormals_to_zero_code](#)()	
[determine_keywords](#)()	A dictionary of values that is made available to the templated.
[get_array_name](#)(var[, access_data])	Get a globally unique name for a `ArrayVariable`.
[translate_expression](#)(expr)	Translate the given expression string into a string in the target language, returns a string.
[translate_one_statement_sequence](#)(statements)	
[translate_statement](#)(statement)	Translate a single line `Statement` into the target language, returns a string.

<div align="right">continues on next page</div>

Table 70 – continued from previous page

translate_statement_sequence(sc_statements, ...)	Translate a sequence of `Statement` into the target language, taking care to declare variables, etc.
translate_to_declarations(read, write, indices)	
translate_to_read_arrays(read, write, indices)	
translate_to_statements(statements, ...)	
translate_to_write_arrays(write)	

Details

flush_denormals

restrict

denormals_to_zero_code()

determine_keywords()
> A dictionary of values that is made available to the templated. This is used for example by the *CPPCode-Generator* to set up all the supporting code

static get_array_name(*var*, *access_data=True*)
> Get a globally unique name for a `ArrayVariable`.

> > **Parameters** **var** : `ArrayVariable`

> > > The variable for which a name should be found.

> > **access_data** : bool, optional

> > > For `DynamicArrayVariable` objects, specifying `True` here means the name for the underlying data is returned. If specifying `False`, the name of object itself is returned (e.g. to allow resizing).

> > **Returns** :

> > **——-** :

> > **name** : str

> > > A uniqe name for `var()`.

translate_expression(*expr*)
> Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)
> Translate a single line `Statement` into the target language, returns a string.

translate_statement_sequence(*sc_statements*, *ve_statements*)
> Translate a sequence of `Statement` into the target language, taking care to declare variables, etc. if necessary.

> Returns a tuple (`scalar_code`, `vector_code`, `kwds`) where `scalar_code` is a list of the lines of code executed before the inner loop, `vector_code` is a list of the lines of code in the inner loop, and `kwds` is a dictionary of values that is made available to the template.

translate_to_declarations(*read*, *write*, *indices*)

> **translate_to_read_arrays**(*read*, *write*, *indices*)
>
> **translate_to_statements**(*statements*, *conditional_write_vars*)
>
> **translate_to_write_arrays**(*write*)

Functions

c_data_type(dtype)	Gives the C language specifier for numpy data types.

c_data_type function

(*Shortest import*: `from brian2.codegen.generators import c_data_type`)

`brian2.codegen.generators.cpp_generator.`**c_data_type**(*dtype*)
> Gives the C language specifier for numpy data types. For example, `numpy.int32` maps to `int32_t` in C.

cython_generator module

Exported members: *CythonCodeGenerator*

Classes

CythonCodeGenerator(*args, **kwds)	Cython code generator

CythonCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import CythonCodeGenerator`)

class `brian2.codegen.generators.cython_generator.`**CythonCodeGenerator**(**args*,
**kwds)

> Bases: *brian2.codegen.generators.base.CodeGenerator*
>
> Cython code generator

Methods

determine_keywords()	A dictionary of values that is made available to the templated.
translate_expression(expr)	Translate the given expression string into a string in the target language, returns a string.
translate_one_statement_sequence(statements)	
translate_statement(statement)	Translate a single line `Statement` into the target language, returns a string.
translate_statement_sequence(sc_statements, ...)	Translate a sequence of `Statement` into the target language, taking care to declare variables, etc.
translate_to_read_arrays(read, indices)	

<div align="right">continues on next page</div>

Table 73 – continued from previous page

translate_to_statements(statements, ...)	
translate_to_write_arrays(write)	

Details

determine_keywords()
> A dictionary of values that is made available to the templated. This is used for example by the `CPPCode-Generator` to set up all the supporting code

translate_expression(*expr*)
> Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)
> Translate a single line `Statement` into the target language, returns a string.

translate_statement_sequence(*sc_statements*, *ve_statements*)
> Translate a sequence of `Statement` into the target language, taking care to declare variables, etc. if necessary.
>
> Returns a tuple (`scalar_code`, `vector_code`, `kwds`) where `scalar_code` is a list of the lines of code executed before the inner loop, `vector_code` is a list of the lines of code in the inner loop, and `kwds` is a dictionary of values that is made available to the template.

translate_to_read_arrays(*read*, *indices*)

translate_to_statements(*statements*, *conditional_write_vars*)

translate_to_write_arrays(*write*)

CythonNodeRenderer([auto_vectorise])	

Methods

CythonNodeRenderer class

(*Shortest import*: from brian2.codegen.generators.cython_generator import CythonN-odeRenderer)

class brian2.codegen.generators.cython_generator.**CythonNodeRenderer**(*auto_vectorise=None*)
> Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

[*render_BinOp*](#)(node)	
[*render_Name*](#)(node)	
[*render_NameConstant*](#)(node)	

Details

render_BinOp(*node*)

render_Name(*node*)

render_NameConstant(*node*)

Functions

[*get_cpp_dtype*](#)(obj)	

get_cpp_dtype function

(*Shortest import*: from brian2.codegen.generators.cython_generator import get_cpp_dtype)

brian2.codegen.generators.cython_generator.**get_cpp_dtype**(*obj*)

[*get_numpy_dtype*](#)(obj)	

get_numpy_dtype function

(*Shortest import*: from brian2.codegen.generators.cython_generator import get_numpy_dtype)

brian2.codegen.generators.cython_generator.**get_numpy_dtype**(*obj*)

`numpy_generator` module

Exported members: [*NumpyCodeGenerator*](#)

Classes

[*NumpyCodeGenerator*](#)(variables, ...[, ...])	Numpy language

NumpyCodeGenerator class

(*Shortest import*: `from brian2.codegen.generators import NumpyCodeGenerator`)

class brian2.codegen.generators.numpy_generator.**NumpyCodeGenerator**(*variables*, *variable_indices*, *owner*, *iterate_all*, *codeobj_class*, *name*, *template_name*, *override_conditional_write=None*, *allows_scalar_write=False*)

Bases: [`brian2.codegen.generators.base.CodeGenerator`](#)

Numpy language

Essentially Python but vectorised.

Methods

[`conditional_write`](#)(line, stmt, variables, ...)	
[`determine_keywords`](#)()	A dictionary of values that is made available to the templated.
[`read_arrays`](#)(read, write, indices, variables, ...)	
[`translate_expression`](#)(expr)	Translate the given expression string into a string in the target language, returns a string.
[`translate_one_statement_sequence`](#)(statements)	
[`translate_statement`](#)(statement)	Translate a single line `Statement` into the target language, returns a string.
[`ufunc_at_vectorisation`](#)(statement, variables, ...)	
[`vectorise_code`](#)(statements, variables, ...[, ...])	
[`write_arrays`](#)(statements, read, write, ...)	

Details

conditional_write(*line*, *stmt*, *variables*, *conditional_write_vars*, *created_vars*)

determine_keywords()
   A dictionary of values that is made available to the templated. This is used for example by the `CPPCodeGenerator` to set up all the supporting code

read_arrays(*read*, *write*, *indices*, *variables*, *variable_indices*)

translate_expression(*expr*)
   Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence (*statements*, *scalar=False*)

translate_statement (*statement*)
Translate a single line `Statement` into the target language, returns a string.

ufunc_at_vectorisation (*statement*, *variables*, *indices*, *conditional_write_vars*, *created_vars*, *used_variables*)

vectorise_code (*statements*, *variables*, *variable_indices*, *index='_idx'*)

write_arrays (*statements*, *read*, *write*, *variables*, *variable_indices*)

[*VectorisationError*]

VectorisationError class

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import Vectorisation-Error`)

class `brian2.codegen.generators.numpy_generator.`**VectorisationError**
Bases: `Exception`

Functions

[*ceil_func*](value)

---

ceil_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import ceil_func`)

`brian2.codegen.generators.numpy_generator.`**ceil_func**(*value*)

[*clip_func*](array, a_min, a_max)

---

clip_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import clip_func`)

`brian2.codegen.generators.numpy_generator.`**clip_func**(*array*, *a_min*, *a_max*)

[*floor_func*](value)

---

floor_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import floor_func`)

`brian2.codegen.generators.numpy_generator.`**`floor_func`**(*value*)

[*int_func*](value)

---

int_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import int_func`)

`brian2.codegen.generators.numpy_generator.`**`int_func`**(*value*)

[*poisson_func*](lam, vectorisation_idx)

---

poisson_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import poisson_func`)

`brian2.codegen.generators.numpy_generator.`**`poisson_func`**(*lam*, *vectorisation_idx*)

[*rand_func*](vectorisation_idx)

---

rand_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import rand_func`)

`brian2.codegen.generators.numpy_generator.`**`rand_func`**(*vectorisation_idx*)

[*randn_func*](vectorisation_idx)

---

randn_func function

(*Shortest import*: `from brian2.codegen.generators.numpy_generator import randn_func`)

`brian2.codegen.generators.numpy_generator.`**`randn_func`**(*vectorisation_idx*)

runtime package

Runtime targets for code generation.

Subpackages

GSLcython_rt package

`GSLcython_rt` module

Module containing the Cython CodeObject for code generation for integration using the ODE solver provided in the GNU Scientific Library (GSL)

Exported members: `GSLCythonCodeObject`, `IntegrationError`

Classes

GSLCompileError

GSLCompileError class

(*Shortest import*: `from brian2.codegen.runtime.GSLcython_rt.GSLcython_rt import GSLCompileError`)

class `brian2.codegen.runtime.GSLcython_rt.GSLcython_rt.`**GSLCompileError**
    Bases: `Exception`

GSLCythonCodeObject(*args, **kw)

<h3 align="center">Methods</h3>

GSLCythonCodeObject class

(*Shortest import*: `from brian2.codegen import GSLCythonCodeObject`)

class `brian2.codegen.runtime.GSLcython_rt.GSLcython_rt.`**GSLCythonCodeObject**(**args*, ***kw*)

    Bases: *brian2.codegen.runtime.cython_rt.cython_rt.CythonCodeObject*

Methods

compile()	

Details

compile()

IntegrationError	Error used to signify that GSL was unable to complete integration (only works for cython)

IntegrationError class

(*Shortest import*: `from brian2.codegen import IntegrationError`)

class `brian2.codegen.runtime.GSLcython_rt.GSLcython_rt.`**`IntegrationError`**
    Bases: `Exception`

    Error used to signify that GSL was unable to complete integration (only works for cython)

cython_rt package

cython_rt module

Exported members: *CythonCodeObject*

Classes

CythonCodeObject(*args, **kw)	Execute code using Cython.

CythonCodeObject class

(*Shortest import*: `from brian2 import CythonCodeObject`)

class `brian2.codegen.runtime.cython_rt.cython_rt.`**`CythonCodeObject`**(*args*, **kw*)
    Bases: *brian2.codegen.runtime.numpy_rt.numpy_rt.NumpyCodeObject*

    Execute code using Cython.

Methods

compile_block(block)	
is_available()	Whether this target for code generation is available.
run_block(block)	
update_namespace()	Update the namespace for this timestep.
variables_to_namespace()	

Details

compile_block(*block*)

classmethod is_available()
> Whether this target for code generation is available. Should use a minimal example to check whether code generation works in general.

run_block(*block*)

update_namespace()
> Update the namespace for this timestep. Should only deal with variables where *the reference* changes every timestep, i.e. where the current reference in `namespace` is not correct.

variables_to_namespace()

extension_manager module

Cython automatic extension builder/manager

Inspired by IPython's Cython cell magics, see: https://github.com/ipython/ipython/blob/master/IPython/extensions/cythonmagic.py

Exported members: `cython_extension_manager`

Classes

CythonExtensionManager()	

Attributes

CythonExtensionManager class

(*Shortest import*: from brian2.codegen.runtime.cython_rt.extension_manager import CythonExtensionManager)

class
brian2.codegen.runtime.cython_rt.extension_manager.**CythonExtensionManager**
> Bases: object

Attributes

[*so_ext*](#)	The extension suffix for compiled modules.

Methods

[*create_extension*](#)(code[, force, name, ...])

Details

so_ext
> The extension suffix for compiled modules.

create_extension(*code*, *force=False*, *name=None*, *define_macros=None*, *include_dirs=None*,
library_dirs=None, *runtime_library_dirs=None*, *extra_compile_args=None*,
extra_link_args=None, *libraries=None*, *compiler=None*, *sources=None*,
owner_name=″)

Functions

[*get_cython_cache_dir*](#)()

get_cython_cache_dir function

(*Shortest import*: from brian2.codegen.runtime.cython_rt.extension_manager import
get_cython_cache_dir)

brian2.codegen.runtime.cython_rt.extension_manager.**get_cython_cache_dir**()

[*get_cython_extensions*](#)()

get_cython_extensions function

(*Shortest import*: from brian2.codegen.runtime.cython_rt.extension_manager import
get_cython_extensions)

brian2.codegen.runtime.cython_rt.extension_manager.**get_cython_extensions**()

[*simplify_path_env_var*](#)(path)

simplify_path_env_var function

(*Shortest import*: `from brian2.codegen.runtime.cython_rt.extension_manager import simplify_path_env_var`)

`brian2.codegen.runtime.cython_rt.extension_manager.`**`simplify_path_env_var`**(*path*)

Objects

[*cython_extension_manager*](#)

---

cython_extension_manager object

(*Shortest import*: `from brian2.codegen.runtime.cython_rt.extension_manager import cython_extension_manager`)

`brian2.codegen.runtime.cython_rt.extension_manager.`**`cython_extension_manager = <brian2.codegen.runtime.cython_rt.extension_manager.CythonExtensionManager object>`**

numpy_rt package

Numpy runtime implementation.

Preferences

Numpy runtime codegen preferences

`codegen.runtime.numpy.discard_units` = `False`

> Whether to change the namespace of user-specifed functions to remove units.

`numpy_rt` module

Module providing [*NumpyCodeObject*](#).

Exported members: [*NumpyCodeObject*](#)

Classes

[*LazyArange*](#)(stop[, start, indices])	A class that can be used as a [arange](#) replacement (with an implied step size of 1) but does not actually create an array of values until necessary.

LazyArange class

(*Shortest import*: `from brian2.codegen.runtime.numpy_rt.numpy_rt import LazyArange`)

class `brian2.codegen.runtime.numpy_rt.numpy_rt.`**`LazyArange`**(*stop*, *start=0*, *indices=None*)
  Bases: `collections.abc.Iterable`

  A class that can be used as a `arange` replacement (with an implied step size of 1) but does not actually create an array of values until necessary. It is somewhat similar to the `range()` function in Python 3, but does not use a generator. It is tailored to a special use case, the `_vectorisation_idx` variable in numpy templates, and not meant for general use. The `_vectorisation_idx` is used for stateless function calls such as `rand()` and for the numpy codegen target determines the number of values produced by such a call. This will often be the number of neurons or synapses, and this class avoids creating a new array of that size at every code object call when all that is needed is the *length* of the array.

Examples

```
>>> from brian2.codegen.runtime.numpy_rt.numpy_rt import LazyArange
>>> ar = LazyArange(10)
>>> len(ar)
10
>>> len(ar[:5])
5
>>> type(ar[:5])
<class 'brian2.codegen.runtime.numpy_rt.numpy_rt.LazyArange'>
>>> ar[5]
5
>>> for value in ar[3:7]:
...     print(value)
...
3
4
5
6
>>> len(ar[np.array([1, 2, 3])])
3
```

NumpyCodeObject(*args, **kw)	Execute code using Numpy

NumpyCodeObject class

(*Shortest import*: `from brian2 import NumpyCodeObject`)

class `brian2.codegen.runtime.numpy_rt.numpy_rt.`**`NumpyCodeObject`**(**args*, ***kw*)
  Bases: *`brian2.codegen.codeobject.CodeObject`*

  Execute code using Numpy

  Default for Brian because it works on all platforms.

Methods

compile_block(block)	

is_available()	Whether this target for code generation is available.
run_block(block)	

update_namespace()	Update the namespace for this timestep.
variables_to_namespace()	

Details

compile_block(*block*)

classmethod is_available()
> Whether this target for code generation is available. Should use a minimal example to check whether code generation works in general.

run_block(*block*)

update_namespace()
> Update the namespace for this timestep. Should only deal with variables where *the reference* changes every timestep, i.e. where the current reference in namespace is not correct.

variables_to_namespace()

6.7.2 core package

Essential Brian modules, in particular base classes for all kinds of brian objects.

Built-in preferences

Core Brian preferences

core.default_float_dtype = float64

> Default dtype for all arrays of scalars (state variables, weights, etc.).

core.default_integer_dtype = int32

> Default dtype for all arrays of integer scalars.

core.outdated_dependency_error = True

> Whether to raise an error for outdated dependencies (True) or just a warning (False).

`base` module

All Brian objects should derive from *`BrianObject`*.

Exported members: *`BrianObject`*, *`BrianObjectException`*

Classes

`BrianObject`(*args, **kw)	All Brian objects derive from this class, defines magic tracking and update.

BrianObject class

(*Shortest import*: `from brian2 import BrianObject`)

class `brian2.core.base.`**`BrianObject`**(*args*, **kw*)
    Bases: *`brian2.core.names.Nameable`*

All Brian objects derive from this class, defines magic tracking and update.

See the documentation for *`Network`* for an explanation of which objects get updated in which order.

    Parameters   **dt** : *`Quantity`*, optional

        The time step to be used for the simulation. Cannot be combined with the *`clock`* argument.

        clock : *`Clock`*, optional

        The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *`defaultclock`* will be used.

        when : str, optional

        In which scheduling slot to simulate the object during a time step. Defaults to `'start'`.

        order : int, optional

        The priority of this object for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

        namespace: dict, optional :

        A dictionary mapping identifier names to objects. If not given, the namespace will be filled in at the time of the call of *`Network.run`*, with either the values from the `namespace` argument of the *`Network.run`* method or from the local context, if no such argument is given.

        name : str, optional

        A unique name for the object - one will be assigned automatically if not provided (of the form `brianobject_1`, etc.).

        Notes :

        —— :

        The set of all `BrianObject` objects is stored in ``BrianObject.__instances__()``. :

Attributes

_clock	The clock used for simulating this object
_creation_stack	A string indicating where this object was created (traceback with any parts of Brian code removed)
_network	Used to remember the Network in which this object has been included before, to raise an error if it is included in a new Network
_scope_current_key	Global key value for ipython cell restrict magic
_scope_key	The scope key is used to determine which objects are collected by magic
active	Whether or not the object should be run.
add_to_magic_network	Whether or not the object should be added to a MagicNetwork.
clock	The Clock determining when the object should be updated.
code_objects	The list of CodeObject contained within the BrianObject.
contained_objects	The list of objects contained within the BrianObject.
invalidates_magic_network	Whether or not MagicNetwork is invalidated when a new BrianObject of this type is added
name	The unique name for this object.
namespace	The group-specific namespace
order	The order in which objects with the same clock and when should be updated
updaters	The list of Updater that define the runtime behaviour of this object.
when	The ID string determining when the object should be updated in Network.run.

Methods

add_dependency(obj)	Add an object to the list of dependencies.
after_run()	Optional method to do work after a run is finished.
before_run(run_namespace)	Optional method to prepare the object before a run.
run()	

Details

_clock
> The clock used for simulating this object

_creation_stack
> A string indicating where this object was created (traceback with any parts of Brian code removed)

_network
> Used to remember the Network in which this object has been included before, to raise an error if it is included in a new Network

_scope_current_key
    Global key value for ipython cell restrict magic

_scope_key
    The scope key is used to determine which objects are collected by magic

active
    Whether or not the object should be run.

    Inactive objects will not have their `update` method called in *Network.run*. Note that setting or unsetting the *active* attribute will set or unset it for all *contained_objects*.

add_to_magic_network
    Whether or not the object should be added to a *MagicNetwork*. Note that all objects in *BrianObject.contained_objects* are automatically added when the parent object is added, therefore e.g. *NeuronGroup* should set *add_to_magic_network* to `True`, but it should not be set for all the dependent objects such as `StateUpdater`

clock
    The *Clock* determining when the object should be updated.

    Note that this cannot be changed after the object is created.

code_objects
    The list of `CodeObject` contained within the *BrianObject*.

    TODO: more details.

    Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.code_objects.extend([A, B])`.

contained_objects
    The list of objects contained within the *BrianObject*.

    When a *BrianObject* is added to a *Network*, its contained objects will be added as well. This allows for compound objects which contain a mini-network structure.

    Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.contained_objects.extend([A, B])`.

invalidates_magic_network
    Whether or not *MagicNetwork* is invalidated when a new *BrianObject* of this type is added

name
    The unique name for this object.

    Used when generating code. Should be an acceptable variable name, i.e. starting with a letter character and followed by alphanumeric characters and _.

namespace
    The group-specific namespace

order
    The order in which objects with the same clock and `when` should be updated

updaters
    The list of `Updater` that define the runtime behaviour of this object.

    TODO: more details.

    Note that this attribute cannot be set directly, you need to modify the underlying list, e.g. `obj.updaters.extend([A, B])`.

when
> The ID string determining when the object should be updated in *Network.run*.

add_dependency(*obj*)
> Add an object to the list of dependencies. Takes care of handling subgroups correctly (i.e., adds its parent object).

> > **Parameters obj** : *BrianObject*

> > > The object that this object depends on.

after_run()
> Optional method to do work after a run is finished.

> Called by *Network.after_run* after the main simulation loop terminated.

before_run(*run_namespace*)
> Optional method to prepare the object before a run.

> Called by *Network.after_run* before the main simulation loop starts.

run()

BrianObjectException(message, brianobj)	High level exception that adds extra Brian-specific information to exceptions

BrianObjectException class

(*Shortest import*: `from brian2 import BrianObjectException`)

class brian2.core.base.**BrianObjectException**(*message*, *brianobj*)
> Bases: Exception

> High level exception that adds extra Brian-specific information to exceptions

> This exception should only be raised at a fairly high level in Brian code to pass information back to the user. It adds extra information about where a *BrianObject* was defined to better enable users to locate the source of problems.

> > **Parameters message** : str

> > > Additional error information to add to the original exception.

> > **brianobj** : BrianObject

> > > The object that caused the error to happen.

> > **original_exception** : Exception

> > > The original exception that was raised.

Functions

brian_object_exception(message, brianobj, ...)	Returns a *BrianObjectException* derived from the original exception.

brian_object_exception function

(*Shortest import*: `from brian2.core.base import brian_object_exception`)

brian2.core.base.**brian_object_exception**(*message*, *brianobj*, *original_exception*)
    Returns a *BrianObjectException* derived from the original exception.

    Creates a new class derived from the class of the original exception and *BrianObjectException*. This allows exception handling code to respond both to the original exception class and *BrianObjectException*.

    See *BrianObjectException* for arguments and notes.

device_override(name)	Decorates a function/method to allow it to be overridden by the current `Device`.

device_override function

(*Shortest import*: `from brian2.core.base import device_override`)

brian2.core.base.**device_override**(*name*)
    Decorates a function/method to allow it to be overridden by the current `Device`.

    The `name` is the function name in the `Device` to use as an override if it exists.

    The returned function has an additional attribute `original_function` which is a reference to the original, undecorated function.

weakproxy_with_fallback(obj)	Attempts to create a `weakproxy` to the object, but falls back to the object if not possible.

weakproxy_with_fallback function

(*Shortest import*: `from brian2.core.base import weakproxy_with_fallback`)

brian2.core.base.**weakproxy_with_fallback**(*obj*)
    Attempts to create a `weakproxy` to the object, but falls back to the object if not possible.

`clocks` module

Clocks for the simulator.

Exported members: *Clock*, *defaultclock*

Classes

Clock(*args, **kw)	An object that holds the simulation time and the time step.

Clock class

(*Shortest import*: `from brian2 import Clock`)

class brian2.core.clocks.**Clock**(**args*, ***kw*)
    Bases: *brian2.groups.group.VariableOwner*

    An object that holds the simulation time and the time step.

        Parameters **dt** : float

            The time step of the simulation as a float

        name : str, optional

            An explicit name, if not specified gives an automatically generated name

Notes

Clocks are run in the same `Network.run` iteration if `t` is the same. The condition for two clocks to be considered as having the same time is `abs(t1-t2)<epsilon*abs(t1)`, a standard test for equality of floating point values. The value of `epsilon` is `1e-14`.

Attributes

dt	The time step of the simulation in seconds.
*dt_*	The time step of the simulation as a float (in seconds)
epsilon_dt	The relative difference for times (in terms of dt) so that they are considered identical.

Methods

set_interval(self, start, end)	Set the start and end time of the simulation.

Details

dt
    The time step of the simulation in seconds.

dt_
    The time step of the simulation as a float (in seconds)

epsilon_dt
    The relative difference for times (in terms of dt) so that they are considered identical.

set_interval(*self*, *start*, *end*)
    Set the start and end time of the simulation.

    Sets the start and end value of the clock precisely if possible (using epsilon) or rounding up if not. This assures that multiple calls to `Network.run` will not re-run the same time step.

Tutorials and examples using this

- Example *COBAHH*

- Example *CUBA*

DefaultClockProxy()	Method proxy to access the defaultclock of the currently active device

DefaultClockProxy class

(*Shortest import*: `from brian2.core.clocks import DefaultClockProxy`)

class `brian2.core.clocks.`**`DefaultClockProxy`**
    Bases: `object`

    Method proxy to access the defaultclock of the currently active device

Functions

check_dt(new_dt, old_dt, target_t)	Check that the target time can be represented equally well with the new dt.

check_dt function

(*Shortest import*: `from brian2.core.clocks import check_dt`)

`brian2.core.clocks.`**`check_dt`** (*new_dt*, *old_dt*, *target_t*)
    Check that the target time can be represented equally well with the new dt.

> **Parameters new_dt** : float
>
> > The new dt value
>
> **old_dt** : float
>
> > The old dt value
>
> **target_t** : float
>
> > The target time

Raises

ValueError If using the new dt value would lead to a difference in the target time of more than *Clock. epsilon_dt* times `new_dt` (by default, 0.01% of the new dt).

Examples

```
>>> from brian2 import *
>>> check_dt(float(17*ms), float(0.1*ms), float(0*ms))  # For t=0s, every dt is␣
↪fine
>>> check_dt(float(0.05*ms), float(0.1*ms), float(10*ms))  # t=10*ms can be␣
↪represented with the new dt
>>> check_dt(float(0.2*ms), float(0.1*ms), float(10.1*ms))  # t=10.1ms cannot be␣
↪represented with dt=0.2ms
Traceback (most recent call last):
...
ValueError: Cannot set dt from 100. us to 200. us, the time 10.1 ms is not a␣
↪multiple of 200. us.
```

Objects

defaultclock	The standard clock, used for objects that do not specify any clock or dt

defaultclock object

(*Shortest import*: `from brian2 import defaultclock`)

brian2.core.clocks.**defaultclock = <brian2.core.clocks.DefaultClockProxy object>**
    The standard clock, used for objects that do not specify any clock or dt

`core_preferences` module

Definitions, documentation, default values and validation functions for core Brian preferences.

Functions

default_float_dtype_validator(dtype)	

default_float_dtype_validator function

(*Shortest import*: `from brian2.core.core_preferences import default_float_dtype_validator`)

brian2.core.core_preferences.**default_float_dtype_validator**(*dtype*)

dtype_repr(dtype)	

dtype_repr function

(*Shortest import*: `from brian2.core.core_preferences import dtype_repr`)

`brian2.core.core_preferences.`**`dtype_repr`**(*dtype*)

`functions` module

Exported members: `DEFAULT_FUNCTIONS`, *`Function`*, *`implementation()`*, *`declare_types()`*

Classes

Function(pyfunc[, sympy_func, arg_units, ...])	An abstract specification of a function that can be used as part of model equations, etc.

Function class

(*Shortest import*: `from brian2 import Function`)

`class` `brian2.core.functions.`**`Function`**(*pyfunc*, *sympy_func=None*, *arg_units=None*, *arg_names=None*, *return_unit=None*, *arg_types=None*, *return_type=None*, *stateless=True*, *auto_vectorise=False*)

   Bases: `object`

   An abstract specification of a function that can be used as part of model equations, etc.

   Parameters **pyfunc** : function

   A Python function that is represented by this *Function* object.

   sympy_func : `sympy.Function`, optional

   A corresponding sympy function (if any). Allows functions to be interpreted by sympy and potentially make simplifications. For example, `sqrt(x**2)` could be replaced by `abs(x)`.

   arg_units : list of *Unit*, optional

   If `pyfunc` does not provide unit information (which typically means that it was not annotated with a *check_units()* decorator), the units of the arguments have to specified explicitly using this parameter.

   return_unit : *Unit* or callable, optional

   Same as for `arg_units`: if `pyfunc` does not provide unit information, this information has to be provided explictly here. `return_unit` can either be a specific *Unit*, if the function always returns the same unit, or a function of the input units, e.g. a "square" function would return the square of its input units, i.e. `return_unit` could be specified as `lambda u: u**2`.

   arg_types : list of str, optional

   Similar to `arg_units`, but gives the type of the argument rather than its unit. In the current version of Brian arguments are specified by one of the following strings: 'boolean', 'integer', 'float', 'any'. If `arg_types` is not specified, 'any' will be assumed. In future versions, a more refined specification may be possible. Note that any argument with a type other than float should have no units. If

   return_type : str, optional

Similar to `return_unit` and `arg_types`. In addition to 'boolean', 'integer' and 'float' you can also use 'highest' which will return the highest type of its arguments. You can also give a function, as for `return_unit`. If the return type is not specified, it is assumed to be 'float'.

stateless : bool, optional

Whether this function does not have an internal state, i.e. if it always returns the same output when called with the same arguments. This is true for mathematical functions but not true for `rand()`, for example. Defaults to `True`.

auto_vectorise : bool, optional

Whether the implementations of this function should get an additional argument (not specified in abstract code) that can be used to determine the number of values that should be returned (for the numpy target), or an index potentially useful for generating deterministic values independent of the order of vectorisation (for all other targets). The main use case are random number functions, e.g. equations refer to `rand()`, but the generate code will actually call `rand(_vectorisation_idx)`. Defaults to `False`.

Notes

If a function should be usable for code generation targets other than Python/numpy, implementations for these target languages have to be added using the `implementation` decorator or using the `add_implementations` function.

Attributes

implementations	Stores implementations for this function in a *FunctionImplementationContainer*

Methods

__call__(*args)	Call self as a function.
is_locally_constant(dt)	Return whether this function (if interpreted as a function of time) should be considered constant over a timestep.

Details

implementations

Stores implementations for this function in a *FunctionImplementationContainer*

__call__(*args*)

Call self as a function.

is_locally_constant(*dt*)

Return whether this function (if interpreted as a function of time) should be considered constant over a timestep. This is most importantly used by *TimedArray* so that linear integration can be used. In its standard implementation, always returns `False`.

Parameters **dt** : float

The length of a timestep (without units).

> **Returns constant** : bool
>
>> Whether the results of this function can be considered constant over one timestep of length
>> `dt`.

FunctionImplementation([name, code, ...])	A simple container object for function implementations.

FunctionImplementation class

(*Shortest import*: `from brian2.core.functions import FunctionImplementation`)

class `brian2.core.functions.`**FunctionImplementation**(*name=None*, *code=None*, *namespace=None*, *dependencies=None*, *availability_check=None*, *dynamic=False*, *compiler_kwds=None*)

> Bases: `object`
>
> A simple container object for function implementations.
>
>> **Parameters name** : str, optional
>>
>>> The name of the function in the target language. Should only be specified if the function
>>> has to be renamed for the target language.
>>
>> **code** : language-dependent, optional
>>
>>> A language dependent argument specifying the implementation in the target language, e.g.
>>> a code string or a dictionary of code strings.
>>
>> **namespace** : dict-like, optional
>>
>>> A dictionary of mappings from names to values that should be added to the namespace of
>>> a `CodeObject` using the function.
>>
>> **dependencies** : dict-like, optional
>>
>>> A mapping of names to *Function* objects, for additional functions needed by this func-
>>> tion.
>>
>> **availability_check** : callable, optional
>>
>>> A function that will be called to check whether the function should be made available (e.g.
>>> depending on whether it is supported by the compiler). The function should do nothing if
>>> the function is available, or raise a `NotImplementedError` with a message explaining
>>> why it isn't.
>>
>> **dynamic** : bool, optional
>>
>>> Whether this `code`/`namespace` is dynamic, i.e. generated for each new context it is
>>> used in. If set to `True`, `code` and `namespace` have to be callable with a *Group* as
>>> an argument and are expected to return the final `code` and `namespace`. Defaults to
>>> `False`.

Methods

[`get_code`](owner)	
[`get_namespace`](owner)	

Details

get_code(*owner*)

get_namespace(*owner*)

[`FunctionImplementationCon-`](FunctionImplementationCon-tainer)[`tainer`](function)	Helper object to store implementations and give access in a dictionary-like fashion, using `CodeGenerator` implementations as a fallback for `CodeObject` implementations.

FunctionImplementationContainer class

(*Shortest import*: `from brian2.core.functions import FunctionImplementationContainer`)

class brian2.core.functions.**FunctionImplementationContainer**(*function*)
  Bases: [`collections.abc.Mapping`](collections.abc.Mapping)

  Helper object to store implementations and give access in a dictionary-like fashion, using `CodeGenerator` implementations as a fallback for `CodeObject` implementations.

Methods

[`add_dynamic_implementation`](target, code[, ...])	Adds an "dynamic implementation" for this function.
[`add_implementation`](target, code[, ...])	
[`add_numpy_implementation`](wrapped_func[, ...])	Add a numpy implementation to a [`Function`](Function).

Details

add_dynamic_implementation(*target*, *code*, *namespace=None*, *dependencies=None*, *availability_check=None*, *name=None*, *compiler_kwds=None*)
  Adds an "dynamic implementation" for this function. [`code`](code) and `namespace` arguments are expected to be callables that will be called in [`Network.before_run`](Network.before_run) with the owner of the `CodeObject` as an argument. This allows to generate code that depends on details of the context it is run in, e.g. the `dt` of a clock.

add_implementation(*target*, *code*, *namespace=None*, *dependencies=None*, *availability_check=None*, *name=None*, *compiler_kwds=None*)

add_numpy_implementation(*wrapped_func*, *dependencies=None*, *discard_units=None*, *compiler_kwds=None*)

Add a numpy implementation to a *Function*.

Parameters **function** : *Function*

The function description for which an implementation should be added.

wrapped_func : callable

The original function (that will be used for the numpy implementation)

dependencies : list of *Function*, optional

A list of functions this function needs.

discard_units : bool, optional

See *implementation()*.

SymbolicConstant(name, sympy_obj, value)	Class for representing constants (e.g.

SymbolicConstant class

(*Shortest import*: `from brian2.core.functions import SymbolicConstant`)

class brian2.core.functions.**SymbolicConstant**(*name*, *sympy_obj*, *value*)

Bases: *brian2.core.variables.Constant*

Class for representing constants (e.g. pi) that are understood by sympy.

exprel(arg)	Represents `(exp(x) - 1)/x`.

exprel class

(*Shortest import*: `from brian2.core.functions import exprel`)

class brian2.core.functions.**exprel**(*arg*)

Bases: `sympy.core.function.Function`

Represents `(exp(x) - 1)/x`.

The benefit of using `exprel(x)` over `(exp(x) - 1)/x` is that the latter is prone to cancellation under finite precision arithmetic when x is close to zero, and cannot be evaluated when x is equal to zero.

Methods

eval(arg)	Returns a canonical form of cls applied to arguments args.
fdiff([argindex])	Returns the first derivative of this function.

Details

classmethod eval(*arg*)
  Returns a canonical form of cls applied to arguments args.

fdiff(*argindex=1*)
  Returns the first derivative of this function.

Tutorials and examples using this

- Example *COBAHH*
- Example *IF_curve_Hodgkin_Huxley*
- Example *advanced/exprel_function*
- Example *advanced/float_32_64_benchmark*
- Example *compartmental/hh_with_spikes*
- Example *compartmental/hodgkin_huxley_1952*
- Example *compartmental/lfp*
- Example *frompapers/Wang_Buszaki_1996*

Functions

declare_types(**types)	Decorator to declare argument and result types for a function

declare_types function

(*Shortest import*: `from brian2 import declare_types`)

`brian2.core.functions.`**`declare_types`**(***types*)
  Decorator to declare argument and result types for a function

  Usage is similar to *check_units()* except that types must be one of {VALID_ARG_TYPES} and the result type must be one of {VALID_RETURN_TYPES}. Unspecified argument types are assumed to be 'all' (i.e. anything is permitted), and an unspecified result type is assumed to be 'float'. Note that the 'highest' option for result type will give the highest type of its argument, e.g. if the arguments were boolean and integer then the result would be integer, if the arguments were integer and float it would be float.

implementation(target[, code, namespace, ...])	A simple decorator to extend user-written Python functions to work with code generation in other languages.

implementation function

(*Shortest import*: `from brian2 import implementation`)

`brian2.core.functions.`**`implementation`**(*target*, *code=None*, *namespace=None*, *dependencies=None*, *discard_units=None*, *name=None*, ***compiler_kwds*)

> A simple decorator to extend user-written Python functions to work with code generation in other languages.
>
> > **Parameters** **target** : str
> >
> > > Name of the code generation target (e.g. `'cython'`) for which to add an implementation.
> >
> > **code** : str or dict-like, optional
> >
> > > What kind of code the target language expects is language-specific, e.g. C++ code allows for a dictionary of code blocks instead of a single string.
> >
> > **namespaces** : dict-like, optional
> >
> > > A namespace dictionary (i.e. a mapping of names to values) that should be added to a `CodeObject` namespace when using this function.
> >
> > **dependencies** : dict-like, optional
> >
> > > A mapping of names to *Function* objects, for additional functions needed by this function.
> >
> > **discard_units: bool, optional** :
> >
> > > Numpy functions can internally make use of the unit system. However, during a simulation run, state variables are passed around as unitless values for efficiency. If `discard_units` is set to `False`, input arguments will have units added to them so that the function can still use units internally (the units will be stripped away from the return value as well). Alternatively, if `discard_units` is set to `True`, the function will receive unitless values as its input. The namespace of the function will be altered to make references to units (e.g. `ms`) refer to the corresponding floating point values so that no unit mismatch errors are raised. Note that this system cannot work in all cases, e.g. it does not work with functions that internally imports values (e.g. does `from brian2 import ms`) or access values with units indirectly (e.g. uses `brian2.ms` instead of `ms`). If no value is given, defaults to the preference setting *codegen.runtime.numpy.discard_units*.
> >
> > **name** : str, optional
> >
> > > The name of the function in the target language. Should only be specified if the function has to be renamed for the target language.
> >
> > **compiler_kwds** : dict, optional
> >
> > > Additional keyword arguments will be transferred to the code generation stage, e.g. for C++-based targets, the code can make use of additional header files by providing a list of strings as the `headers` argument.

Notes

While it is in principle possible to provide a numpy implementation as an argument for this decorator, this is normally not necessary – the numpy implementation should be provided in the decorated function.

If this decorator is used with other decorators such as *check_units()* or *declare_types()*, it should be the uppermost decorator (that is, the last one to be applied).

Examples

Sample usage:

```
@implementation('cpp','''
            #include<math.h>
            inline double usersin(double x)
            {
                return sin(x);
            }
            ''')
def usersin(x):
    return sin(x)
```

timestep(t, dt)	Converts a given time to an integer time step.

timestep function

(*Shortest import*: `from brian2.core.functions import timestep`)

brian2.core.functions.**timestep**(*t*, *dt*)

Converts a given time to an integer time step. This function slightly shifts the time before dividing it by `dt` to make sure that multiples of `dt` do not end up in the preceding time step due to floating point issues. This function is used in the refractoriness calculation.

New in version 2.1.3.

> **Parameters** **t** : np.ndarray, float, Quantity
>
> > The time to convert.
>
> **dt** : float or Quantity
>
> > The length of a simulation time step.
>
> **Returns** **ts** : np.ndarray, np.int64
>
> > The time step corresponding to the given time.

Notes

This function cannot handle infinity values, use big values instead (e.g. a *NeuronGroup* will use `-1e4*second` as the value of the `lastspike` variable for neurons that never spiked).

`magic` module

Exported members: *MagicNetwork*, *magic_network*, *MagicError*, *run()*, *stop()*, *collect()*, *store()*, *restore()*, *start_scope()*

Classes

MagicError	Error that is raised when something goes wrong in *Mag-icNetwork*

MagicError class

(*Shortest import*: `from brian2 import MagicError`)

class `brian2.core.magic.`**MagicError**
  Bases: `Exception`

  Error that is raised when something goes wrong in *MagicNetwork*

  See notes to *MagicNetwork* for more details.

MagicNetwork(*args, **kw)	*Network* that automatically adds all Brian objects

MagicNetwork class

(*Shortest import*: `from brian2 import MagicNetwork`)

class `brian2.core.magic.`**MagicNetwork**(*args*, **kw*)
  Bases: *brian2.core.network.Network*

  Network that automatically adds all Brian objects

  In order to avoid bugs, this class will occasionally raise *MagicError* when the intent of the user is not clear. See the notes below for more details on this point. If you persistently see this error, then Brian is not able to safely guess what you intend to do, and you should use a *Network* object and call *Network.run* explicitly.

  Note that this class cannot be instantiated by the user, there can be only one instance *magic_network* of *Mag-icNetwork*.

  See also:

  Network, *collect()*, *run()*, *stop()*, *store()*, *restore()*

Notes

All Brian objects that are visible at the point of the `run()` call will be included in the network. This class is designed to work in the following two major use cases:

1. You create a collection of Brian objects, and call `run()` to run the simulation. Subsequently, you may call `run()` again to run it again for a further duration. In this case, the `Network.t` time will start at 0 and for the second call to `run()` will continue from the end of the previous run.

2. You have a loop in which at each iteration, you create some Brian objects and run a simulation using them. In this case, time is reset to 0 for each call to `run()`.

In any other case, you will have to explicitly create a `Network` object yourself and call `Network.run` on this object. Brian has a built in system to guess which of the cases above applies and behave correctly. When it is not possible to safely guess which case you are in, it raises `MagicError`. The rules for this guessing system are explained below.

If a simulation consists only of objects that have not been run, it will assume that you want to start a new simulation. If a simulation only consists of objects that have been simulated in the previous `run()` call, it will continue that simulation at the previous time.

If neither of these two situations apply, i.e., the network consists of a mix of previously run objects and new objects, an error will be raised.

In these checks, "non-invalidating" objects (i.e. objects that have `BrianObject.invalidates_magic_network` set to False) are ignored, e.g. creating new monitors is always possible.

Methods

`add`(*objs)	You cannot add objects directly to `MagicNetwork`
`after_run`()	
`check_dependencies`()	
`get_states`([units, format, subexpressions, ...])	See `Network.get_states`.
`remove`(*objs)	You cannot remove objects directly from `MagicNetwork`
`restore`([name, filename, ...])	See `Network.restore`.
`run`(duration[, report, report_period, ...])	Runs the simulation for the given duration.
`set_states`(values[, units, format, level])	See `Network.set_states`.
`store`([name, filename, level])	See `Network.store`.

Details

add(*objs*)
    You cannot add objects directly to `MagicNetwork`

after_run()

check_dependencies()

get_states(*units=True*, *format='dict'*, *subexpressions=False*, *level=0*)
    See `Network.get_states`.

remove (**objs*)

> You cannot remove objects directly from *MagicNetwork*

restore (*name='default'*, *filename=None*, *restore_random_state=False*, *level=0*)

> See *Network.restore*.

run (*duration*, *report=None*, *report_period=60 * second*, *namespace=None*, *level=0*)

> Runs the simulation for the given duration.

> **Parameters duration** : *Quantity*

>> The amount of simulation time to run for.

>> **report** : {None, 'text', 'stdout', 'stderr', function}, optional

>> How to report the progress of the simulation. If None, do not report progress. If 'text' or 'stdout' is specified, print the progress to stdout. If 'stderr' is specified, print the progress to stderr. Alternatively, you can specify a callback callable(elapsed, complete, duration) which will be passed the amount of time elapsed as a *Quantity*, the fraction complete from 0.0 to 1.0 and the total duration of the simulation (in biological time). The function will always be called at the beginning and the end (i.e. for fractions 0.0 and 1.0), regardless of the report_period.

>> **report_period** : *Quantity*

>> How frequently (in real time) to report progress.

>> **namespace** : dict-like, optional

>> A namespace that will be used in addition to the group-specific namespaces (if defined). If not specified, the locals and globals around the run function will be used.

>> **profile** : bool, optional

>> Whether to record profiling information (see *Network.profiling_info*). Defaults to None (which will use the value set by set_device, if any).

>> **level** : int, optional

>> How deep to go up the stack frame to look for the locals/global (see namespace argument). Only used by run functions that call this run function, e.g. *MagicNetwork.run* to adjust for the additional nesting.

Notes

The simulation can be stopped by calling *Network.stop* or the global *stop()* function.

set_states (*values*, *units=True*, *format='dict'*, *level=0*)

> See *Network.set_states*.

store (*name='default'*, *filename=None*, *level=0*)

> See *Network.store*.

Functions

collect([level])	Return the list of *BrianObject*s that will be simulated if *run()* is called.

collect function

(*Shortest import*: `from brian2 import collect`)

`brian2.core.magic.`**`collect`**(*level=0*)

Return the list of [`BrianObject`](#)s that will be simulated if [`run()`](#) is called.

> **Parameters** **level** : int, optional
>
> > How much further up to go in the stack to find the objects. Needs only to be specified if [`collect()`](#) is called as part of a function and should be increased by 1 for every level of nesting. Defaults to 0.
>
> **Returns** **objects** : set of [`BrianObject`](#)
>
> > The objects that will be simulated.

[`get_objects_in_namespace`](#)(level)	Get all the objects in the current namespace that derive from [`BrianObject`](#).

get_objects_in_namespace function

(*Shortest import*: `from brian2.core.magic import get_objects_in_namespace`)

`brian2.core.magic.`**`get_objects_in_namespace`**(*level*)

Get all the objects in the current namespace that derive from [`BrianObject`](#). Used to determine the objects for the [`MagicNetwork`](#).

> **Parameters** **level** : int, optional
>
> > How far to go back to get the locals/globals. Each function/method call should add `1` to this argument, functions/method with a decorator have to add `2`.
>
> **Returns** **objects** : set
>
> > A set with weak references to the [`BrianObject`](#)s in the namespace.

[`restore`](#)([name, filename, restore_random_state])	Restore the state of the network and all included objects.

restore function

(*Shortest import*: `from brian2 import restore`)

`brian2.core.magic.`**`restore`**(*name='default'*, *filename=None*, *restore_random_state=False*)

Restore the state of the network and all included objects.

> **Parameters** **name** : str, optional
>
> > The name of the snapshot to restore, if not specified uses `'default'`.
>
> **filename** : str, optional
>
> > The name of the file from where the state should be restored. If not specified, it is expected that the state exist in memory (i.e. [`Network.store`](#) was previously called without the `filename` argument).
>
> **restore_random_state** : bool, optional

Whether to restore the state of the random number generator. If set to `True`, going back to an earlier state of the simulation will continue exactly where it left off, even if the simulation is stochastic. If set to `False` (the default), random numbers are independent between runs (except for explicitly set random seeds), regardless of whether *store()*/*restore()* has been used or not. Note that this also restores numpy's random number generator (since it is used internally by Brian), but it does *not* restore Python's builtin random number generator in the `random` module.

See Also :

—— :

Network.restore :

run(duration[, report, report_period, ...])	Runs a simulation with all "visible" Brian objects for the given duration.

run function

(*Shortest import*: `from brian2 import run`)

`brian2.core.magic.`**run**(*duration*, *report=None*, *report_period=10 * second*, *namespace=None*, *level=0*)

Runs a simulation with all "visible" Brian objects for the given duration. Calls *collect()* to gather all the objects, the simulation can be stopped by calling the global *stop()* function.

In order to avoid bugs, this function will occasionally raise *MagicError* when the intent of the user is not clear. See the notes to *MagicNetwork* for more details on this point. If you persistently see this error, then Brian is not able to safely guess what you intend to do, and you should use a *Network* object and call *Network.run* explicitly.

Parameters **duration** : *Quantity*

The amount of simulation time to run for. If the network consists of new objects since the last time *run()* was called, the start time will be reset to 0. If *run()* is called twice or more without changing the set of objects, the second and subsequent runs will start from the end time of the previous run. To explicitly reset the time to 0, do `magic_network.t = 0*second`.

report : {None, 'stdout', 'stderr', 'graphical', function}, optional

How to report the progress of the simulation. If None, do not report progress. If stdout or stderr is specified, print the progress to stdout or stderr. If graphical, Tkinter is used to show a graphical progress bar. Alternatively, you can specify a callback `function(elapsed, complete)` which will be passed the amount of time elapsed (in seconds) and the fraction complete from 0 to 1.

report_period : *Quantity*

How frequently (in real time) to report progress.

profile : bool, optional

Whether to record profiling information (see *Network.profiling_info*). Defaults to None (which will use the value set by `set_device`, if any).

namespace : dict-like, optional

A namespace in which objects which do not define their own namespace will be run. If not namespace is given, the locals and globals around the run function will be used.

> **level** : int, optional
>
>> How deep to go down the stack frame to look for the locals/global (see `namespace` argument). Only necessary under particular circumstances, e.g. when calling the run function as part of a function call or lambda expression. This is used in tests, e.g.: `assert_raises(MagicError, lambda: run(1*ms, level=3))`.

> ### Raises
>
> *`MagicError`* Error raised when it was not possible for Brian to safely guess the intended use. See *`MagicNet-work`* for more details.
>
> **See also:**
>
> *`Network.run`*, *`MagicNetwork`*, *`collect()`*, *`start_scope()`*, *`stop()`*

`start_scope`()	Starts a new scope for magic functions

start_scope function

(*Shortest import*: `from brian2 import start_scope`)

`brian2.core.magic.`**`start_scope`**()
    Starts a new scope for magic functions

   All objects created before this call will no longer be automatically included by the magic functions such as *`run()`*.

`stop`()	Stops all running simulations.

stop function

(*Shortest import*: `from brian2 import stop`)

`brian2.core.magic.`**`stop`**()
    Stops all running simulations.

> **See also:**
>
> *`Network.stop`*, *`run()`*, `reinit`

`store`([name, filename])	Store the state of the network and all included objects.

store function

(*Shortest import*: `from brian2 import store`)

`brian2.core.magic.`**`store`**(*name='default'*, *filename=None*)
    Store the state of the network and all included objects.

> **Parameters** **name** : str, optional
>
>> A name for the snapshot, if not specified uses `'default'`.
>
> **filename** : str, optional

A filename where the state should be stored. If not specified, the state will be stored in memory.

See also:

Network.store

Objects

magic_network	Automatically constructed *MagicNetwork* of all Brian objects

magic_network object

(*Shortest import*: from brian2 import magic_network)

brian2.core.magic.**magic_network = MagicNetwork()**
Automatically constructed *MagicNetwork* of all Brian objects

names module

Exported members: *Nameable*

Classes

Nameable(*args, **kw)	Base class to find a unique name for an object

Nameable class

(*Shortest import*: from brian2 import Nameable)

class brian2.core.names.**Nameable**(*args*, **kw*)
Bases: *brian2.core.tracking.Trackable*

Base class to find a unique name for an object

If you specify a name explicitly, and it has already been taken, a `ValueError` is raised. You can also specify a name with a wildcard asterisk in the end, i.e. in the form `'name*'`. It will then try `name` first but if this is already specified, it will try name_1, name__2`, etc. This is the default mechanism used by most core objects in Brian, e.g. *NeuronGroup* uses a default name of `'neurongroup*'`.

    Parameters **name** : str

An name for the object, possibly ending in * to specify that variants of this name should be tried if the name (without the asterisk) is already taken. If (and only if) the name for this object has already been set, it is also possible to call the initialiser with None for the *name* argument. This situation can arise when a class derives from multiple classes that derive themselves from *Nameable* (e.g. *Group* and *CodeRunner*) and their initialisers are called explicitely.

Raises

ValueError If the name is already taken.

Attributes

id	A unique id for this object.
name	The unique name for this object.

Methods

assign_id()	Assign a new id to this object.

Details

id
A unique id for this object.

In contrast to names, which may be reused, the id stays unique. This is used in the dependency checking to not have to deal with the chore of comparing weak references, weak proxies and strong references.

name
The unique name for this object.

Used when generating code. Should be an acceptable variable name, i.e. starting with a letter character and followed by alphanumeric characters and _.

assign_id()
Assign a new id to this object. Under most circumstances, this method should only be called once at the creation of the object to generate a unique id. In the case of the *MagicNetwork*, however, the id should change when a new, independent set of objects is simulated.

Functions

find_name(name)	

find_name function

(*Shortest import*: `from brian2.core.names import find_name`)

brian2.core.names.**find_name**(*name*)

`namespace` **module**

Implementation of the namespace system, used to resolve the identifiers in model equations of *NeuronGroup* and *Synapses*

Exported members: *get_local_namespace()*, DEFAULT_FUNCTIONS, DEFAULT_UNITS, DE-FAULT_CONSTANTS

Functions

get_local_namespace(level)	Get the surrounding namespace.

get_local_namespace **function**

(*Shortest import*: `from brian2 import get_local_namespace`)

brian2.core.namespace.**get_local_namespace**(*level*)

Get the surrounding namespace.

> **Parameters level** : int, optional
>
> > How far to go back to get the locals/globals. Each function/method call should add `1` to this argument, functions/method with a decorator have to add `2`.
>
> **Returns namespace** : dict
>
> > The locals and globals at the given depth of the stack frame.

`network` **module**

Module defining the *Network* object, the basis of all simulation runs.

Preferences

Network preferences

core.network.default_schedule = ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']

> Default schedule used for networks that don't specify a schedule.

Exported members: *Network*, *profiling_summary()*, *scheduling_summary()*

Classes

Network(*objs[, name])	The main simulation controller in Brian

Network class

(*Shortest import*: `from brian2 import Network`)

class brian2.core.network.**Network**(**objs*, *name='network*'*)
   Bases: *brian2.core.names.Nameable*

   The main simulation controller in Brian

   Network handles the running of a simulation. It contains a set of Brian objects that are added with *add*. The *run* method actually runs the simulation. The main run loop, determining which objects get called in what order is described in detail in the notes below. The objects in the *Network* are accesible via their names, e.g. `net['neurongroup']` would return the *NeuronGroup* with this name.

   > **Parameters** **objs** : (*BrianObject*, container), optional
   >
   > > A list of objects to be added to the *Network* immediately, see *add*.
   >
   > **name** : str, optional
   >
   > > An explicit name, if not specified gives an automatically generated name

   See also:

   MagicNetwork, *run()*, *stop()*

   ### Notes

   The main run loop performs the following steps:

   1. Prepare the objects if necessary, see `prepare`.

   2. Determine the end time of the simulation as `t`+``duration``.

   3. Determine which set of clocks to update. This will be the clock with the smallest value of `t`. If there are several with the same value, then all objects with these clocks will be updated simultaneously. Set `t` to the clock time.

   4. If the `t` value of these clocks is past the end time of the simulation, stop running. If the *Network.stop* method or the *stop()* function have been called, stop running. Set `t` to the end time of the simulation.

   5. For each object whose `clock` is set to one of the clocks from the previous steps, call the `update` method. This method will not be called if the `active` flag is set to `False`. The order in which the objects are called is described below.

   6. Increase `Clock.t` by *Clock.dt* for each of the clocks and return to step 2.

   The order in which the objects are updated in step 4 is determined by the *Network.schedule* and the objects `when` and `order` attributes. The *schedule* is a list of string names. Each `when` attribute should be one of these strings, and the objects will be updated in the order determined by the schedule. The default schedule is `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`. In addition to the names provided in the schedule, automatic names starting with `before_` and `after_` can be used. That means that all objects with `when=='before_start'` will be updated first, then those with `when=='start'`, `when=='after_start'`, `when=='before_groups'`, `when=='groups'` and so forth. If several objects have the same `when` attribute, then the order is determined by the `order` attribute (lower first).

Attributes

_stored_state	Stored state of objects (store/restore)
objects	The set of objects in the Network, should not normally be modified directly.
profiling_info	The time spent in executing the various CodeObject s.
schedule	List of when slots in the order they will be updated, can be modified.
sorted_objects	The sorted objects of this network in the order defined by the schedule.
t	Current simulation time in seconds (Quantity)
t_	Current time as a float

Methods

add(*objs)	Add objects to the Network
after_run()	
before_run(namespace)	Prepares the Network for a run.
check_dependencies()	
get_profiling_info()	The only reason this is not directly implemented in profiling_info is to allow devices (e.g.
get_states([units, format, subexpressions, ...])	Return a copy of the current state variable values of objects in the network.
remove(*objs)	Remove an object or sequence of objects from a Network.
restore([name, filename, restore_random_state])	Retore the state of the network and all included objects.
run(duration[, report, report_period, ...])	Runs the simulation for the given duration.
scheduling_summary()	Return a SchedulingSummary object, representing the scheduling information for all objects included in the network.
set_states(values[, units, format, level])	Set the state variables of objects in the network.
stop()	Stops the network from running, this is reset the next time Network.run is called.
store([name, filename])	Store the state of the network and all included objects.

Details

_stored_state
    Stored state of objects (store/restore)

objects
    The set of objects in the Network, should not normally be modified directly. Note that in a *MagicNetwork*, this attribute only contains the objects during a run: it is filled in *before_run* and emptied in *after_run*

profiling_info
    The time spent in executing the various CodeObject s.

A list of (name, time) tuples, containing the name of the CodeObject and the total execution time for simulations of this object (as a *Quantity* with unit second). The list is sorted descending with execution time.

Profiling has to be activated using the profile keyword in *run()* or *Network.run*.

schedule

List of when slots in the order they will be updated, can be modified.

See notes on scheduling in *Network*. Note that additional when slots can be added, but the schedule should contain at least all of the names in the default schedule: ['start', 'groups', 'thresholds', 'synapses', 'resets', 'end'].

The schedule can also be set to None, resetting it to the default schedule set by the *core.network.default_schedule* preference.

sorted_objects

The sorted objects of this network in the order defined by the schedule.

Objects are sorted first by their when attribute, and secondly by the order attribute. The order of the when attribute is defined by the schedule. In addition to the slot names defined in the schedule, automatic slot names starting with before_ and after_ can be used (e.g. the slots ['groups', 'thresholds'] allow to use ['before_groups', 'groups', 'after_groups', 'before_thresholds', 'thresholds', 'after_thresholds']).

Final ties are resolved using the objects' names, leading to an arbitrary but deterministic sorting.

t

Current simulation time in seconds (*Quantity*)

t_

Current time as a float

add(**objs*)

Add objects to the *Network*

> **Parameters objs** : (*BrianObject*, container)
>
>> The *BrianObject* or container of Brian objects to be added. Specify multiple objects, or lists (or other containers) of objects. Containers will be added recursively. If the container is a dict then it will add the values from the dictionary but not the keys. If you want to add the keys, do add(objs.keys()).

after_run()

before_run(*namespace*)

Prepares the *Network* for a run.

Objects in the *Network* are sorted into the correct running order, and their *BrianObject.before_run* methods are called.

> **Parameters run_namespace** : dict-like, optional
>
>> A namespace in which objects which do not define their own namespace will be run.

check_dependencies()

get_profiling_info()

The only reason this is not directly implemented in *profiling_info* is to allow devices (e.g. CPPStandaloneDevice) to overwrite this.

get_states (*units=True*, *format='dict'*, *subexpressions=False*, *read_only_variables=True*, *level=0*)

> Return a copy of the current state variable values of objects in the network.. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

> > **Parameters** **vars** : list of str, optional

> > > The names of the variables to extract. If not specified, extract all state variables (except for internal variables, i.e. names that start with `'_'`). If the `subexpressions` argument is `True`, the current values of all subexpressions are returned as well.

> > **units** : bool, optional

> > > Whether to include the physical units in the return value. Defaults to `True`.

> > **format** : str, optional

> > > The output format. Defaults to `'dict'`.

> > **subexpressions: bool, optional** :

> > > Whether to return subexpressions when no list of variable names is given. Defaults to `False`. This argument is ignored if an explicit list of variable names is given in `vars`.

> > **read_only_variables** : bool, optional

> > > Whether to return read-only variables (e.g. the number of neurons, the time, etc.). Setting it to `False` will assure that the returned state can later be used with *set_states*. Defaults to `True`.

> > **level** : int, optional

> > > How much higher to go up the stack to resolve external variables. Only relevant if extracting subexpressions that refer to external variables.

> > **Returns** **values** : dict

> > > A dictionary mapping object names to the state variables of that object, in the specified `format`.

> **See also:**

> *VariableOwner.get_states*

remove (**objs*)

> Remove an object or sequence of objects from a *Network*.

> > **Parameters** **objs** : (*BrianObject*, container)

> > > The *BrianObject* or container of Brian objects to be removed. Specify multiple objects, or lists (or other containers) of objects. Containers will be removed recursively.

restore (*name='default'*, *filename=None*, *restore_random_state=False*)

> Retore the state of the network and all included objects.

> > **Parameters** **name** : str, optional

> > > The name of the snapshot to restore, if not specified uses `'default'`.

> > **filename** : str, optional

> > > The name of the file from where the state should be restored. If not specified, it is expected that the state exist in memory (i.e. *Network.store* was previously called without the `filename` argument).

> > **restore_random_state** : bool, optional

Whether to restore the state of the random number generator. If set to `True`, going back to an earlier state of the simulation will continue exactly where it left off, even if the simulation is stochastic. If set to `False` (the default), random numbers are independent between runs (except for explicitly set random seeds), regardless of whether *store()*/*restore()* has been used or not. Note that this also restores numpy's random number generator (since it is used internally by Brian), but it does *not* restore Python's builtin random number generator in the `random` module.

run (*duration*, *report=None*, *report_period=60 * second*, *namespace=None*, *level=0*)

Runs the simulation for the given duration.

>   **Parameters duration** : *Quantity*

>   The amount of simulation time to run for.

>   **report** : {None, 'text', 'stdout', 'stderr', function}, optional

>   How to report the progress of the simulation. If `None`, do not report progress. If `'text'` or `'stdout'` is specified, print the progress to stdout. If `'stderr'` is specified, print the progress to stderr. Alternatively, you can specify a callback `callable(elapsed, complete, duration)` which will be passed the amount of time elapsed as a *Quantity*, the fraction complete from 0.0 to 1.0 and the total duration of the simulation (in biological time). The function will always be called at the beginning and the end (i.e. for fractions 0.0 and 1.0), regardless of the `report_period`.

>   **report_period** : *Quantity*

>   How frequently (in real time) to report progress.

>   **namespace** : dict-like, optional

>   A namespace that will be used in addition to the group-specific namespaces (if defined). If not specified, the locals and globals around the run function will be used.

>   **profile** : bool, optional

>   Whether to record profiling information (see *Network.profiling_info*). Defaults to `None` (which will use the value set by `set_device`, if any).

>   **level** : int, optional

>   How deep to go up the stack frame to look for the locals/global (see `namespace` argument). Only used by run functions that call this run function, e.g. *MagicNetwork.run* to adjust for the additional nesting.

Notes

The simulation can be stopped by calling *Network.stop* or the global *stop()* function.

scheduling_summary ()

Return a *SchedulingSummary* object, representing the scheduling information for all objects included in the network.

>   **Returns summary** : *SchedulingSummary*

>   Object representing the scheduling information.

set_states (*values*, *units=True*, *format='dict'*, *level=0*)

Set the state variables of objects in the network.

>   **Parameters values** : dict

>   A dictionary mapping object names to objects of `format`, setting the states of this object.

> **units** : bool, optional
>
>> Whether the `values` include physical units. Defaults to `True`.
>
> **format** : str, optional
>
>> The format of `values`. Defaults to `'dict'`
>
> **level** : int, optional
>
>> How much higher to go up the stack to _resolve external variables. Only relevant when using string expressions to set values.

See also:

`Group.set_states`

stop()
  Stops the network from running, this is reset the next time `Network.run` is called.

store(*name='default'*, *filename=None*)
  Store the state of the network and all included objects.

> **Parameters name** : str, optional
>
>> A name for the snapshot, if not specified uses `'default'`.
>
> **filename** : str, optional
>
>> A filename where the state should be stored. If not specified, the state will be stored in memory.

Notes

The state stored to disk can be restored with the `Network.restore` function. Note that it will only restore the *internal state* of all the objects (including undelivered spikes) – the objects have to exist already and they need to have the same name as when they were stored. Equations, thresholds, etc. are *not* stored – this is therefore not a general mechanism for object serialization. Also, the format of the file is not guaranteed to work across platforms or versions. If you are interested in storing the state of a network for documentation or analysis purposes use `Network.get_states` instead.

Tutorials and examples using this

- Tutorial *3-intro-to-brian-simulations*
- Example *IF_curve_Hodgkin_Huxley*
- Example *IF_curve_LIF*
- Example *advanced/compare_GSL_to_conventional*
- Example *advanced/stochastic_odes*
- Example *frompapers/Brunel_Wang_2001*
- Example *multiprocessing/02_using_standalone*
- Example *multiprocessing/03_standalone_joblib*
- Example *standalone/standalone_multiplerun*

ProfilingSummary(net[, show])	Class to nicely display the results of profiling.

ProfilingSummary class

(*Shortest import*: `from brian2.core.network import ProfilingSummary`)

class `brian2.core.network.`**`ProfilingSummary`**(*net*, *show=None*)

Bases: `object`

Class to nicely display the results of profiling. Objects of this class are returned by *profiling_summary()*.

> **Parameters net** : *Network*
>
> > The *Network* object to profile.
>
> **show** : int, optional
>
> > The number of results to show (the longest results will be shown). If not specified, all results will be shown.

See also:

Network.profiling_info

SchedulingSummary(objects)	Object representing the schedule that is used to simulate the objects in a network.

SchedulingSummary class

(*Shortest import*: `from brian2.core.network import SchedulingSummary`)

class `brian2.core.network.`**`SchedulingSummary`**(*objects*)

Bases: `object`

Object representing the schedule that is used to simulate the objects in a network. Objects of this type are returned by *scheduling_summary()*, they should not be created manually by the user.

> **Parameters objects** : list of *BrianObject*
>
> > The sorted list of objects that are simulated by the network.

TextReport(stream)	Helper object to report simulation progress in *Network.run*.

TextReport class

(*Shortest import*: `from brian2.core.network import TextReport`)

class `brian2.core.network.`**`TextReport`**(*stream*)

Bases: `object`

Helper object to report simulation progress in *Network.run*.

> **Parameters stream** : file
>
> > The stream to write to, commonly `sys.stdout` or `sys.stderr`.

Methods

__call__(elapsed, completed, start, duration)	Call self as a function.

Details

__call__(*elapsed*, *completed*, *start*, *duration*)
    Call self as a function.

Functions

profiling_summary([net, show])	Returns a *ProfilingSummary* of the profiling info for a run.

profiling_summary function

(*Shortest import*: `from brian2 import profiling_summary`)

`brian2.core.network.`**`profiling_summary`**(*net=None*, *show=None*)
    Returns a *ProfilingSummary* of the profiling info for a run. This object can be transformed to a string explicitly but on an interactive console simply calling *profiling_summary()* is enough since it will automatically convert the *ProfilingSummary* object.

> **Parameters** **net** : {*Network*, None} optional
>
> > The *Network* object to profile, or *magic_network* if not specified.
>
> **show** : int
>
> > The number of results to show (the longest results will be shown). If not specified, all results will be shown.

schedule_propagation_offset([net])	Returns the minimal time difference for a post-synaptic effect after a spike.

schedule_propagation_offset function

(*Shortest import*: `from brian2.core.network import schedule_propagation_offset`)

`brian2.core.network.`**`schedule_propagation_offset`**(*net=None*)
    Returns the minimal time difference for a post-synaptic effect after a spike. With the default schedule, this time difference is 0, since the `thresholds` slot precedes the `synapses` slot. For the GeNN device, however, a post-synaptic effect will occur in the following time step, this function therefore returns one `dt`.

> **Parameters** **net** : *Network*
>
> > The network to check (uses the magic network if not specified).
>
> **Returns** **offset** : *Quantity*
>
> > The minimum spike propagation delay: `0*ms` for the standard schedule but `dt` for schedules where `synapses` precedes `thresholds`.

Notes

This function always returns `0*ms` or `defaultclock.dt` – no attempt is made to deal with other clocks.

scheduling_summary([net])	Returns a *SchedulingSummary* object, representing the scheduling information for all objects included in the given *Network* (or the "magic" network, if none is specified).

scheduling_summary function

(*Shortest import*: `from brian2 import scheduling_summary`)

`brian2.core.network.`**`scheduling_summary`**(*net=None*)

Returns a *SchedulingSummary* object, representing the scheduling information for all objects included in the given *Network* (or the "magic" network, if none is specified). The returned objects can be printed or converted to a string to give an ASCII table representation of the schedule. In a Jupyter notebook, the output can be displayed as a HTML table.

> **Parameters net** : *Network*, optional
>
>> The network for which the scheduling information should be displayed. Defaults to the "magic" network.
>
> **Returns summary** : *SchedulingSummary*
>
>> An object that represents the scheduling information.

`operations` module

Exported members: *NetworkOperation*, *network_operation()*

Classes

NetworkOperation(*args, **kw)	Object with function that is called every time step.

NetworkOperation class

(*Shortest import*: `from brian2 import NetworkOperation`)

class `brian2.core.operations.`**`NetworkOperation`**(**args, **kw*)

> Bases: *brian2.core.base.BrianObject*

Object with function that is called every time step.

> **Parameters function** : function
>
>> The function to call every time step, should take either no arguments in which case it is called as `function()` or one argument, in which case it is called with the current *Clock* time (*Quantity*).
>
> **dt** : *Quantity*, optional
>
>> The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : [*Clock*], optional

> The update clock to be used. If neither a clock, nor the dt argument is specified, the [*defaultclock*] will be used.

when : str, optional

> In which scheduling slot to execute the operation during a time step. Defaults to 'start'.

order : int, optional

> The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

See also:

[*network_operation()*], [*Network*], [*BrianObject*]

Attributes

[*function*]	The function to be called each time step

Methods

[*run*]()	

Details

function
> The function to be called each time step

run()

Functions

[*network_operation*]([when])	Decorator to make a function get called every time step of a simulation.

network_operation function

(*Shortest import*: from brian2 import network_operation)

brian2.core.operations.**network_operation**(*when=None*)
> Decorator to make a function get called every time step of a simulation.

> The function being decorated should either have no arguments, or a single argument which will be called with the current time t.

>> **Parameters dt** : [*Quantity*], optional

>>> The time step to be used for the simulation. Cannot be combined with the clock argument.

> **clock** : *Clock*, optional
>
>> The update clock to be used. If neither a clock, nor the dt argument is specified, the *defaultclock* will be used.
>
> **when** : str, optional
>
>> In which scheduling slot to execute the operation during a time step. Defaults to 'start'.
>
> **order** : int, optional
>
>> The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

See also:

NetworkOperation, *Network*, *BrianObject*

Notes

Converts the function into a *NetworkOperation*.

If using the form:

```
@network_operations(when='end')
def f():
    ...
```

Then the arguments to network_operation must be keyword arguments.

Examples

Print something each time step: >>> from brian2 import * >>> @network_operation ... def f(): ... print('something') ... >>> net = Network(f)

Print the time each time step:

```
>>> @network_operation
... def f(t):
...    print('The time is', t)
...
>>> net = Network(f)
```

Specify a dt, etc.:

```
>>> @network_operation(dt=0.5*ms, when='end')
... def f():
...    print('This will happen at the end of each timestep.')
...
>>> net = Network(f)
```

preferences module

Brian global preferences are stored as attributes of a *BrianGlobalPreferences* object `prefs`.

Exported members: *PreferenceError*, *BrianPreference*, *prefs*, *brian_prefs*

Classes

BrianGlobalPreferences()	Class of the `prefs` object.

BrianGlobalPreferences class

(*Shortest import*: `from brian2.core.preferences import BrianGlobalPreferences`)

class brian2.core.preferences.**BrianGlobalPreferences**

> Bases: `collections.abc.MutableMapping`
>
> Class of the `prefs` object.
>
> Used for getting/setting/validating/registering preference values. All preferences must be registered via *register_preferences*. To get or set a preference, you can either use a dictionary-based or an attribute-based interface:
>
> ```
> prefs['core.default_float_dtype'] = float32
> prefs.core.default_float_dtype = float32
> ```
>
> Preferences can be read from files, see *load_preferences* and *read_preference_file*. Note that *load_preferences* is called automatically when Brian has finished importing.

Attributes

as_file	Get a Brian preference doc file format string for the current preferences
defaults_as_file	Get a Brian preference doc file format string for the default preferences
toplevel_categories	The toplevel preference categories

Methods

check_all_validated()	Checks that all preferences that have been set have been validated.
do_validation()	Validates preferences that have not yet been validated.
eval_pref(value)	Evaluate a string preference in the units namespace
get_documentation([basename, link_targets])	Generates a string documenting all preferences with the given `basename`.
load_preferences()	Load all the preference files, but do not validate them.
read_preference_file(file)	Reads a Brian preferences file
register_preferences(prefbasename, ...)	Registers a set of preference names, docs and validation functions.
reset_to_defaults()	Resets the parameters to their default values.

Details

as_file
  Get a Brian preference doc file format string for the current preferences

defaults_as_file
  Get a Brian preference doc file format string for the default preferences

toplevel_categories
  The toplevel preference categories

check_all_validated()
  Checks that all preferences that have been set have been validated.

  Logs a warning if not. Should be called by *Network.run* or other key Brian functions.

do_validation()
  Validates preferences that have not yet been validated.

eval_pref(*value*)
  Evaluate a string preference in the units namespace

get_documentation(*basename=None*, *link_targets=True*)
  Generates a string documenting all preferences with the given `basename`. If no `basename` is given, all preferences are documented.

load_preferences()
  Load all the preference files, but do not validate them.

  Preference files are read in the following order:

  1. `~/.brian/user_preferences` from the user's home directory

  2. `./brian_preferences` from the current directory

  Files that are missing are ignored. Preferences read at each step override preferences from previous steps.

  See also:

  read_preference_file

read_preference_file(*file*)
  Reads a Brian preferences file

  The file format for Brian preferences is a plain text file of the form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
e = 3
```

  Blank and comment lines are ignored, all others should be of one of the following two forms:

```
key = value
[section]
```

  `eval` is called on the values, so strings should be written as, e.g. `'3'` rather than `3`. The eval is called with all unit names available. Within a section, the section name is prepended to the key. So in the above example, it would give the following unvalidated dictionary:

```
{'a.b.c': 1,
 'a.b.d': 2,
 'a.b.e': 3,
 }
```

> **Parameters file** : file, str
>
> > The file object or filename of the preference file.

register_preferences(*prefbasename*, *prefbasedoc*, ***prefs*)

> Registers a set of preference names, docs and validation functions.
>
> > **Parameters prefbasename** : str
> >
> > > The base name of the preference.
> >
> > **prefbasedoc** : str
> >
> > > Documentation for this base name
> >
> > ****prefs** : dict of (name, *BrianPreference*) pairs
> >
> > > The preference names to be defined. The full preference name will be prefbasename.
> > > name, and the *BrianPreference* value is used to define the default value, docs, and
> > > validation function.

Raises

PreferenceError If the base name is already registered.

> See also:
>
> *BrianPreference*

reset_to_defaults()

> Resets the parameters to their default values.

BrianGlobalPreferencesView(basename, all_prefs)	A class allowing for accessing preferences in a subcategory.

BrianGlobalPreferencesView class

(*Shortest import*: from brian2.core.preferences import BrianGlobalPreferencesView)

class brian2.core.preferences.**BrianGlobalPreferencesView**(*basename*, *all_prefs*)

> Bases: collections.abc.MutableMapping

A class allowing for accessing preferences in a subcategory. It forwards requests to *BrianGlobalPreferences* and provides documentation and autocompletion support for all preferences in the given category. This object is used to allow accessing preferences via attributes of the *prefs* object.

> **Parameters basename** : str
>
> > The name of the preference category. Has to correspond to a key in
> > BrianGlobalPreferences.pref_register.
>
> **all_prefs** : *BrianGlobalPreferences*

A reference to the main object storing the preferences.

BrianPreference(default, docs[, validator, ...])	Used for defining a Brian preference.

BrianPreference class

(*Shortest import*: `from brian2 import BrianPreference`)

class `brian2.core.preferences.`**`BrianPreference`**(*default*, *docs*, *validator=None*, *representor=<built-in function repr>*)

Bases: `object`

Used for defining a Brian preference.

> **Parameters default** : object
>
> > The default value.
>
> **docs** : str
>
> > Documentation for the preference value.
>
> **validator** : func
>
> > A function that True or False depending on whether the preference value is valid or not. If not specified, uses the *DefaultValidator* for the default value provided (check if the class is the same, and for *Quantity* objects, whether the units are consistent).
>
> **representor** : func
>
> > A function that returns a string representation of a valid preference value that can be passed to `eval`. By default, uses `repr` which works in almost all cases.

DefaultValidator(value)	Default preference validator

DefaultValidator class

(*Shortest import*: `from brian2.core.preferences import DefaultValidator`)

class `brian2.core.preferences.`**`DefaultValidator`**(*value*)

Bases: `object`

Default preference validator

Used by *BrianPreference* as the default validator if none is given. First checks if the provided value is of the same class as the default value, and then if the default is a *Quantity*, checks that the units match.

Methods

__call__(value)	Call self as a function.

Details

__call__(*value*)
Call self as a function.

ErrorRaiser()

ErrorRaiser class

(*Shortest import*: `from brian2.core.preferences import ErrorRaiser`)

class `brian2.core.preferences.`**ErrorRaiser**
Bases: `object`

PreferenceError	Exception relating to the Brian preferences system.

PreferenceError class

(*Shortest import*: `from brian2 import PreferenceError`)

class `brian2.core.preferences.`**PreferenceError**
Bases: `Exception`

Exception relating to the Brian preferences system.

Functions

check_preference_name(name)	Make sure that a preference name is valid.

check_preference_name function

(*Shortest import*: `from brian2.core.preferences import check_preference_name`)

`brian2.core.preferences.`**check_preference_name**(*name*)
Make sure that a preference name is valid. This currently checks that the name does not contain illegal characters and does not clash with method names such as "keys" or "items".

> **Parameters** **name** : str
>
> > The name to check.

Raises

PreferenceError In case the name is invalid.

parse_preference_name(name)	Split a preference name into a base and end name.

parse_preference_name function

(*Shortest import*: `from brian2.core.preferences import parse_preference_name`)

brian2.core.preferences.**parse_preference_name**(*name*)

Split a preference name into a base and end name.

> **Parameters name** : str
>
>> The full name of the preference.
>
> **Returns basename** : str
>
>> The first part of the name up to the final `.`.
>
>> **endname** : str
>
>> The last part of the name from the final `.` onwards.

Examples

```
>>> parse_preference_name('core.default_float_dtype')
('core', 'default_float_dtype')
>>> parse_preference_name('codegen.cpp.compiler')
('codegen.cpp', 'compiler')
```

Objects

brian_prefs

brian_prefs object

(*Shortest import*: `from brian2 import brian_prefs`)

brian2.core.preferences.**brian_prefs = <brian2.core.preferences.ErrorRaiser object>**

prefs	Preference categories:

prefs object

(*Shortest import*: `from brian2 import prefs`)

`brian2.core.preferences.`**`prefs = <BrianGlobalPreferences with top-level categories: "logging", "devices", "GSL", "codegen", "core", "legacy">`**
    Preference categories:

    ** logging ** Logging system preferences

    ** devices ** Device preferences

    ** GSL ** Directory containing GSL code

    ** codegen ** Code generation preferences

    ** core ** Core Brian preferences

    ** legacy ** Preferences to enable legacy behaviour

`spikesource` module

Exported members: *[SpikeSource](#)*

Classes

[SpikeSource](#)()	A source of spikes.

SpikeSource class

(*Shortest import*: `from brian2 import SpikeSource`)

class `brian2.core.spikesource.`**`SpikeSource`**
    Bases: [`object`](#)

    A source of spikes.

    An object that can be used as a source of spikes for objects such as *[SpikeMonitor](#)*, *[Synapses](#)*, etc.

    The defining properties of *[SpikeSource](#)* are that it should have:

    • A length that can be extracted with `len(obj)`, where the maximum spike index possible is `len(obj)-1`.

    • An attribute *[spikes](#)*, an array of ints each from 0 to `len(obj)-1` with no repeats (but possibly not in sorted order). This should be updated each time step.

    • A *[clock](#)* attribute, this will be used as the default clock for objects with this as a source.

    spikes
        An array of ints, each from 0 to `len(obj)-1` with no repeats (but possibly not in sorted order). Updated each time step.

    clock
        The clock on which the spikes will be updated.

`tracking` **module**

Exported members: *[Trackable](#)*

Classes

[InstanceFollower](#)()	Keep track of all instances of classes derived from *[Trackable](#)*

InstanceFollower class

(*Shortest import*: `from brian2.core.tracking import InstanceFollower`)

class `brian2.core.tracking.`**`InstanceFollower`**
    Bases: [object](#)

    Keep track of all instances of classes derived from *[Trackable](#)*

    The variable `__instancesets__` is a dictionary with keys which are class objects, and values which are *[In-](#)* *[stanceTrackerSet](#)*, so `__instanceset__[cls]` is a set tracking all of the instances of class `cls` (or a subclass).

Methods

[add](#)(value)	
[get](#)(cls)	

Details

add (*value*)

get (*cls*)

[InstanceTrackerSet](#)	A [set](#) of [weakref.ref](#) to all existing objects of a certain class.

InstanceTrackerSet class

(*Shortest import*: `from brian2.core.tracking import InstanceTrackerSet`)

class `brian2.core.tracking.`**`InstanceTrackerSet`**
    Bases: [set](#)

    A [set](#) of [weakref.ref](#) to all existing objects of a certain class.

    Should not normally be directly used.

Methods

add(value)	Adds a `weakref.ref` to the `value`
remove(value)	Removes the `value` (which should be a weakref) if it is in the set

Details

add(*value*)
    Adds a `weakref.ref` to the `value`

remove(*value*)
    Removes the `value` (which should be a weakref) if it is in the set

    Sometimes the value will have been removed from the set by `clear`, so we ignore `KeyError` in this case.

Trackable(*args, **kw)	Classes derived from this will have their instances tracked.

Trackable class

(*Shortest import*: `from brian2 import Trackable`)

class brian2.core.tracking.**Trackable**(**args*, ***kw*)
    Bases: `object`

    Classes derived from this will have their instances tracked.

    The `classmethod __instances__()` will return an *InstanceTrackerSet* of the instances of that class, and its subclasses.

variables module

Classes used to specify the type of a function, variable or common sub-expression.

Exported members: *Variable*, *Constant*, *ArrayVariable*, *DynamicArrayVariable*, *Subexpression*, *AuxiliaryVariable*, *VariableView*, *Variables*, *LinkedVariable*, *linked_var()*

Classes

ArrayVariable(name, owner, size, device[, ...])	An object providing information about a model variable stored in an array (for example, all state variables).

ArrayVariable class

(*Shortest import*: `from brian2.core.variables import ArrayVariable`)

class `brian2.core.variables.`**`ArrayVariable`**(*name*, *owner*, *size*, *device*, *dimensions=Dimension()*, *dtype=None*, *constant=False*, *scalar=False*, *read_only=False*, *dynamic=False*, *unique=False*)

Bases: *`brian2.core.variables.Variable`*

An object providing information about a model variable stored in an array (for example, all state variables). Most of the time *`Variables.add_array`* should be used instead of instantiating this class directly.

> **Parameters** **name** : 'str'
>
> > The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable `v` of a *`NeuronGroup`* is known as `v_post` in a `Synapse` connecting to the group).
>
> **dimensions** : `Dimension`, optional
>
> > The physical dimensions of the variable
>
> **owner** : *`Nameable`*
>
> > The object that "owns" this variable, e.g. the *`NeuronGroup`* or *`Synapses`* object that declares the variable in its model equations.
>
> **size** : int
>
> > The size of the array
>
> **device** : `Device`
>
> > The device responsible for the memory access.
>
> **dtype** : `dtype`, optional
>
> > The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.
>
> **constant** : bool, optional
>
> > Whether the variable's value is constant during a run. Defaults to `False`.
>
> **scalar** : bool, optional
>
> > Whether this array is a 1-element array that should be treated like a scalar (e.g. for a single delay value across synapses). Defaults to `False`.
>
> **read_only** : bool, optional
>
> > Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.
>
> **unique** : bool, optional
>
> > Whether the values in this array are all unique. This information is only important for variables used as indices and does not have to reflect the actual contents of the array but only the possibility of non-uniqueness (e.g. synaptic indices are always unique but the corresponding pre- and post-synaptic indices are not). Defaults to `False`.

Attributes

conditional_write	Another variable, on which the write is conditioned (e.g.
device	The `Device` responsible for memory access.
size	The size of this variable.
unique	Wether all values in this arrays are necessarily unique (only relevant for index variables).

Methods

get_addressable_value(name, group)	Get the value (without units) of this variable in a form that can be indexed in the context of a group.
get_addressable_value_with_unit(name, group)	Get the value (with units) of this variable in a form that can be indexed in the context of a group.
get_len()	Get the length of the value associated with the variable or `0` for a scalar variable.
get_value()	Return the value associated with the variable (without units).
set_conditional_write(var)	
set_value(value)	Set the value associated with the variable.

Details

conditional_write
> Another variable, on which the write is conditioned (e.g. a variable denoting the absence of refractoriness)

device
> The `Device` responsible for memory access.

size
> The size of this variable.

unique
> Wether all values in this arrays are necessarily unique (only relevant for index variables).

get_addressable_value(*name*, *group*)
> Get the value (without units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable x is accessed in a synapse S as S.x_post, the synaptic indexing scheme can be used.

> > **Parameters name** : str
> >
> > > The name of the variable
> >
> > **group** : *Group*
> >
> > > The group providing the context for the indexing. Note that this group is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.
> >
> > **Returns variable** : object
> >
> > > The variable in an indexable form (without units).

get_addressable_value_with_unit(*name*, *group*)

    Get the value (with units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable x is accessed in a synapse S as S.x_post, the synaptic indexing scheme can be used.

        Parameters name : str

            The name of the variable

        group : *Group*

            The group providing the context for the indexing. Note that this group is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.

        Returns variable : object

            The variable in an indexable form (with units).

get_len()

    Get the length of the value associated with the variable or 0 for a scalar variable.

get_value()

    Return the value associated with the variable (without units). This is the way variables are accessed in generated code.

set_conditional_write(*var*)

set_value(*value*)

    Set the value associated with the variable.

AuxiliaryVariable(name[, dimensions, dtype, ...])	Variable description for an auxiliary variable (most likely one that is added automatically to abstract code, e.g.

AuxiliaryVariable class

(*Shortest import*: from brian2.core.variables import AuxiliaryVariable)

class brian2.core.variables.**AuxiliaryVariable**(*name*, *dimensions=Dimension()*, *dtype=None*, *scalar=False*)

    Bases: *brian2.core.variables.Variable*

Variable description for an auxiliary variable (most likely one that is added automatically to abstract code, e.g. _cond for a threshold condition), specifying its type and unit for code generation. Most of the time *Variables.add_auxiliary_variable* should be used instead of instantiating this class directly.

        Parameters name : str

            The name of the variable

        dimensions : Dimension, optional

            The physical dimensions of the variable.

        dtype : *dtype*, optional

            The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

        scalar : bool, optional

Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

Methods

[`get_value`]()	Return the value associated with the variable (without units).

Details

get_value()

Return the value associated with the variable (without units). This is the way variables are accessed in generated code.

[`Constant`](name, value[, dimensions, owner])	A scalar constant (e.g.

Constant class

(*Shortest import*: `from brian2.core.variables import Constant`)

class brian2.core.variables.**Constant**(*name*, *value*, *dimensions=Dimension()*, *owner=None*)

Bases: [`brian2.core.variables.Variable`]()

A scalar constant (e.g. the number of neurons `N`). Information such as the dtype or whether this variable is a boolean are directly derived from the [`value`](). Most of the time [`Variables.add_constant`]() should be used instead of instantiating this class directly.

> **Parameters   name** : str
>
>> The name of the variable
>
> **dimensions** : `Dimension`, optional
>
>> The physical dimensions of the variable. Note that the variable itself (as referenced by value) should never have units attached.
>
> **value: reference to the variable value** :
>
>> The value of the constant.
>
> **owner** : [`Nameable`](), optional
>
>> The object that "owns" this variable, for constants that belong to a specific group, e.g. the `N` constant for a [`NeuronGroup`](). External constants will have `None` (the default value).

Attributes

value	The constant's value

Methods

get_value()	Return the value associated with the variable (without units).

Details

value
> The constant's value

get_value()
> Return the value associated with the variable (without units). This is the way variables are accessed in generated code.

DynamicArrayVariable(name, owner, size, device)	An object providing information about a model variable stored in a dynamic array (used in *Synapses*).

DynamicArrayVariable class

(*Shortest import*: `from brian2.core.variables import DynamicArrayVariable`)

class brian2.core.variables.**DynamicArrayVariable**(*name*, *owner*, *size*, *device*, *dimensions=Dimension( )*, *dtype=None*, *constant=False*, *needs_reference_update=False*, *resize_along_first=False*, *scalar=False*, *read_only=False*, *unique=False*)

> Bases: *brian2.core.variables.ArrayVariable*

An object providing information about a model variable stored in a dynamic array (used in *Synapses*). Most of the time *Variables.add_dynamic_array* should be used instead of instantiating this class directly.

> **Parameters   name** : 'str'
>
> > The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable v of a *NeuronGroup* is known as v_post in a Synapse connecting to the group).
>
> **dimensions** : `Dimension`, optional
>
> > The physical dimensions of the variable.
>
> **owner** : *Nameable*
>
> > The object that "owns" this variable, e.g. the *NeuronGroup* or *Synapses* object that declares the variable in its model equations.
>
> **size** : int or tuple of int
>
> > The (initial) size of the variable.

device : `Device`

> The device responsible for the memory access.

dtype : `dtype`, optional

> The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

> Whether the variable's value is constant during a run. Defaults to `False`.

needs_reference_update : bool, optional

> Whether the code objects need a new reference to the underlying data at every time step. This should be set if the size of the array can be changed by other code objects. Defaults to `False`.

scalar : bool, optional

> Whether this array is a 1-element array that should be treated like a scalar (e.g. for a single delay value across synapses). Defaults to `False`.

read_only : bool, optional

> Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

unique : bool, optional

> Whether the values in this array are all unique. This information is only important for variables used as indices and does not have to reflect the actual contents of the array but only the possibility of non-uniqueness (e.g. synaptic indices are always unique but the corresponding pre- and post-synaptic indices are not). Defaults to `False`.

Attributes

dimensions	

ndim	The number of dimensions
needs_reference_update	Whether this variable needs an update of the reference to the underlying data whenever it is passed to a code object
resize_along_first	Whether this array will be only resized along the first dimension

Methods

resize(new_size)	Resize the dynamic array.

Details

dimensions

ndim
>   The number of dimensions

needs_reference_update
>   Whether this variable needs an update of the reference to the underlying data whenever it is passed to a code object

resize_along_first
>   Whether this array will be only resized along the first dimension

resize(*new_size*)
>   Resize the dynamic array. Calls `self.device.resize` to do the actual resizing.
>
>   > **Parameters new_size** : int or tuple of int
>   >
>   > > The new size.

| *LinkedVariable*(group, name, variable[, index]) | A simple helper class to make linking variables explicit. |

LinkedVariable class

(*Shortest import*: `from brian2.core.variables import LinkedVariable`)

class brian2.core.variables.**LinkedVariable**(*group*, *name*, *variable*, *index=None*)
>   Bases: `object`

A simple helper class to make linking variables explicit. Users should use *linked_var()* instead.

>   **Parameters group** : *Group*
>
>   > The group through which the `variable` is accessed (not necessarily the same as `variable.owner`.
>
>   **name** : str
>
>   > **The name of variable in group (not necessarily the same as** `variable.name`).
>
>   **variable** : *Variable*
>
>   > The variable that should be linked.
>
>   **index** : str or `ndarray`, optional
>
>   > An indexing array (or the name of a state variable), providing a mapping from the entries in the link source to the link target.

| *Subexpression*(name, owner, expr, device[, ...]) | An object providing information about a named subexpression in a model. |

Subexpression class

(*Shortest import*: `from brian2.core.variables import Subexpression`)

class brian2.core.variables.**Subexpression**(*name*, *owner*, *expr*, *device*, *dimensions=Dimension()*,
dtype=None, *scalar=False*)

    Bases: *brian2.core.variables.Variable*

An object providing information about a named subexpression in a model. Most of the time *Variables.add_subexpression* should be used instead of instantiating this class directly.

    Parameters **name** : str

        The name of the subexpression.

    dimensions : `Dimension`, optional

        The physical dimensions of the subexpression.

    owner : *Group*

        The group to which the expression refers.

    expr : str

        The subexpression itself.

    device : `Device`

        The device responsible for the memory access.

    dtype : `dtype`, optional

        The dtype used for the expression. Defaults to *core.default_float_dtype*.

    scalar: bool, optional :

        Whether this is an expression only referring to scalar variables. Defaults to `False`

Attributes

device	The `Device` responsible for memory access
expr	The expression defining the subexpression
identifiers	The identifiers used in the expression

Methods

get_addressable_value(name, group)	Get the value (without units) of this variable in a form that can be indexed in the context of a group.
get_addressable_value_with_unit(name, group)	Get the value (with units) of this variable in a form that can be indexed in the context of a group.

Details

device
> The `Device` responsible for memory access

expr
> The expression defining the subexpression

identifiers
> The identifiers used in the expression

get_addressable_value(*name*, *group*)
> Get the value (without units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable `x` is accessed in a synapse S as `S.x_post`, the synaptic indexing scheme can be used.
>
> > **Parameters name** : str
> >
> > > The name of the variable
> >
> > **group** : *Group*
> >
> > > The group providing the context for the indexing. Note that this `group` is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.
> >
> > **Returns variable** : object
> >
> > > The variable in an indexable form (without units).

get_addressable_value_with_unit(*name*, *group*)
> Get the value (with units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable `x` is accessed in a synapse S as `S.x_post`, the synaptic indexing scheme can be used.
>
> > **Parameters name** : str
> >
> > > The name of the variable
> >
> > **group** : *Group*
> >
> > > The group providing the context for the indexing. Note that this `group` is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.
> >
> > **Returns variable** : object
> >
> > > The variable in an indexable form (with units).

Variable(name[, dimensions, owner, dtype, ...])	An object providing information about model variables (including implicit variables such as `t` or `xi`).

Variable class

(*Shortest import*: `from brian2.core.variables import Variable`)

class brian2.core.variables.**Variable**(*name*, *dimensions=Dimension()*, *owner=None*, *dtype=None*, *scalar=False*, *constant=False*, *read_only=False*, *dynamic=False*, *array=False*)

   Bases: *brian2.utils.caching.CacheKey*

   An object providing information about model variables (including implicit variables such as `t` or `xi`). This class should never be instantiated outside of testing code, use one of its subclasses instead.

   Parameters   name : 'str'

   > The name of the variable. Note that this refers to the *original* name in the owning group. The same variable may be known under other names in other groups (e.g. the variable `v` of a *NeuronGroup* is known as `v_post` in a `Synapse` connecting to the group).

   dimensions : `Dimension`, optional

   > The physical dimensions of the variable.

   owner : *Nameable*, optional

   > The object that "owns" this variable, e.g. the *NeuronGroup* or *Synapses* object that declares the variable in its model equations. Defaults to `None` (the value used for *Variable* objects without an owner, e.g. external *Constant*s).

   dtype : `dtype`, optional

   > The dtype used for storing the variable. Defaults to the preference `core.default_scalar.dtype`.

   scalar : bool, optional

   > Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

   constant: bool, optional :

   > Whether the value of this variable can change during a run. Defaults to `False`.

   read_only : bool, optional

   > Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user (this is used for example for the variable `N`, the number of neurons in a group). Defaults to `False`.

   array : bool, optional

   > Whether this variable is an array. Allows for simpler check than testing `isinstance(var, ArrayVariable)`. Defaults to `False`.

Attributes

array	Whether the variable is an array
constant	Whether the variable is constant during a run
dim	The variable's dimensions.
dtype	The dtype used for storing the variable.
dtype_str	String representation of the numpy dtype
dynamic	Whether the variable is dynamically sized (only for non-scalars)
is_boolean	
is_integer	
name	The variable's name.
owner	The *Group* to which this variable belongs.
read_only	Whether the variable is read-only
scalar	Whether the variable is a scalar
unit	The *Unit* of this variable

Methods

get_addressable_value(name, group)	Get the value (without units) of this variable in a form that can be indexed in the context of a group.
get_addressable_value_with_unit(name, group)	Get the value (with units) of this variable in a form that can be indexed in the context of a group.
get_len()	Get the length of the value associated with the variable or `0` for a scalar variable.
get_value()	Return the value associated with the variable (without units).
get_value_with_unit()	Return the value associated with the variable (with units).
set_value(value)	Set the value associated with the variable.

Details

array
> Whether the variable is an array

constant
> Whether the variable is constant during a run

dim
> The variable's dimensions.

dtype
> The dtype used for storing the variable.

dtype_str
> String representation of the numpy dtype

dynamic
> Whether the variable is dynamically sized (only for non-scalars)

is_boolean

is_integer

name
> The variable's name.

owner
> The *Group* to which this variable belongs.

read_only
> Whether the variable is read-only

scalar
> Whether the variable is a scalar

unit
> The *Unit* of this variable

get_addressable_value(*name*, *group*)
> Get the value (without units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable x is accessed in a synapse S as S.x_post, the synaptic indexing scheme can be used.

> **Parameters name** : str

> > The name of the variable

> **group** : *Group*

> > The group providing the context for the indexing. Note that this group is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.

> **Returns variable** : object

> > The variable in an indexable form (without units).

get_addressable_value_with_unit(*name*, *group*)
> Get the value (with units) of this variable in a form that can be indexed in the context of a group. For example, if a postsynaptic variable x is accessed in a synapse S as S.x_post, the synaptic indexing scheme can be used.

> **Parameters name** : str

> > The name of the variable

> **group** : *Group*

> > The group providing the context for the indexing. Note that this group is not necessarily the same as *Variable.owner*: a variable owned by a *NeuronGroup* can be indexed in a different way if accessed via a *Synapses* object.

> **Returns variable** : object

> > The variable in an indexable form (with units).

get_len()
> Get the length of the value associated with the variable or 0 for a scalar variable.

get_value()
> Return the value associated with the variable (without units). This is the way variables are accessed in generated code.

get_value_with_unit()
> Return the value associated with the variable (with units).

set_value(*value*)
> Set the value associated with the variable.

VariableView(name, variable, group[, dimensions])	A view on a variable that allows to treat it as an numpy array while allowing special indexing (e.g.

VariableView class

(*Shortest import*: from brian2.core.variables import VariableView)

class brian2.core.variables.**VariableView**(*name*, *variable*, *group*, *dimensions=None*)
> Bases: object

A view on a variable that allows to treat it as an numpy array while allowing special indexing (e.g. with strings) in the context of a *Group*.

> **Parameters** **name** : str
>
> > The name of the variable (not necessarily the same as variable.name).
>
> **variable** : *Variable*
>
> > The variable description.
>
> **group** : *Group*
>
> > The group through which the variable is accessed (not necessarily the same as variable.owner).
>
> **dimensions** : Dimension, optional
>
> > The physical dimensions to be used for the variable, should be *None* when a variable is accessed without units (e.g. when accessing G.var_).

Attributes

dtype	
shape	
unit	The *Unit* of this variable

Methods

get_item(item[, level, namespace])	Get the value of this variable.
get_subexpression_with_index_array(item, ...)	
get_with_expression(code, run_namespace)	Gets a variable using a string expression.
get_with_index_array(item)	

Table 201 – continued from previous page

set_item(item, value[, level, namespace])	Set this variable.
set_with_expression(item, code, run_namespace)	Sets a variable using a string expression.
set_with_expression_conditional(cond, code, ...)	Sets a variable using a string expression and string condition.
set_with_index_array(item, value, check_units)	

Details

dtype

shape

unit
    The *Unit* of this variable

get_item(*item*, *level=0*, *namespace=None*)
    Get the value of this variable. Called by __getitem__.

>   **Parameters item** : slice, `ndarray` or string
>
>>   The index for the setting operation
>
>   **level** : int, optional
>
>>   How much farther to go up in the stack to find the implicit namespace (if used, see `run_namespace`).
>
>   **namespace** : dict-like, optional
>
>>   An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

get_subexpression_with_index_array(*item*, *run_namespace*)

get_with_expression(*code*, *run_namespace*)
    Gets a variable using a string expression. Is called by *VariableView.get_item* for statements such as `print(G.v['g_syn > 0'])`.

>   **Parameters code** : str
>
>>   An expression that states a condition for elements that should be selected. Can contain references to indices, such as `i` or `j` and to state variables. For example: `'i>3 and v>0*mV'`.
>
>   **run_namespace** : dict-like
>
>>   An additional namespace that is used for variable lookup (either an explicitly defined namespace or one taken from the local context).

get_with_index_array(*item*)

set_item(*item*, *value*, *level=0*, *namespace=None*)
    Set this variable. This function is called by __setitem__ but there is also a situation where it should be called directly: if the context for string-based expressions is higher up in the stack, this function allows to set the `level` argument accordingly.

>   **Parameters item** : slice, `ndarray` or string
>
>>   The index for the setting operation

> **value** : *Quantity*, ndarray or number
>
> > The value for the setting operation
>
> **level** : int, optional
>
> > How much farther to go up in the stack to find the implicit namespace (if used, see run_namespace).
>
> **namespace** : dict-like, optional
>
> > An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

set_with_expression(*item*, *code*, *run_namespace*, *check_units=True*)
> Sets a variable using a string expression. Is called by *VariableView.set_item* for statements such as S.var[:, :] = 'exp(-abs(i-j)/space_constant)*nS'

> > **Parameters item** : ndarray
> >
> > > The indices for the variable (in the context of this group).
> >
> > **code** : str
> >
> > > The code that should be executed to set the variable values. Can contain references to indices, such as i or j
> >
> > **run_namespace** : dict-like, optional
> >
> > > An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).
> >
> > **check_units** : bool, optional
> >
> > > Whether to check the units of the expression.
> >
> > **run_namespace** : dict-like, optional
> >
> > > An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

set_with_expression_conditional(*cond*, *code*, *run_namespace*, *check_units=True*)
> Sets a variable using a string expression and string condition. Is called by *VariableView.set_item* for statements such as S.var['i!=j'] = 'exp(-abs(i-j)/space_constant)*nS'

> > **Parameters cond** : str
> >
> > > The string condition for which the variables should be set.
> >
> > **code** : str
> >
> > > The code that should be executed to set the variable values.
> >
> > **run_namespace** : dict-like, optional
> >
> > > An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).
> >
> > **check_units** : bool, optional
> >
> > > Whether to check the units of the expression.

set_with_index_array(*item*, *value*, *check_units*)

Variables(owner[, default_index])	A container class for storing *Variable* objects.

Variables class

(*Shortest import*: `from brian2.core.variables import Variables`)

class brian2.core.variables.**Variables**(*owner*, *default_index='_idx'*)
    Bases: `collections.abc.Mapping`

    A container class for storing *Variable* objects. Instances of this class are used as the `Group.variables` attribute and can be accessed as (read-only) dictionaries.

        Parameters owner : *Nameable*

            The object (typically a *Group*) "owning" the variables.

        default_index : str, optional

            The index to use for the variables (only relevant for *ArrayVariable* and *DynamicArrayVariable*). Defaults to `'_idx'`.

Attributes

indices	A dictionary given the index name for every array name
owner	A reference to the *Group* owning these variables

Methods

add_arange(name, size[, start, dtype, ...])	Add an array, initialized with a range of integers.
add_array(name, size[, dimensions, values, ...])	Add an array (initialized with zeros).
add_arrays(names, size[, dimensions, dtype, ...])	Adds several arrays (initialized with zeros) with the same attributes (size, units, etc.).
add_auxiliary_variable(name[, dimensions, ...])	Add an auxiliary variable (most likely one that is added automatically to abstract code, e.g.
add_constant(name, value[, dimensions])	Add a scalar constant (e.g.
add_dynamic_array(name, size[, dimensions, ...])	Add a dynamic array.
add_object(name, obj)	Add an arbitrary Python object.
add_reference(name, group[, varname, index])	Add a reference to a variable defined somewhere else (possibly under a different name).
add_references(group, varnames[, index])	Add all *Variable* objects from a name to *Variable* mapping with the same name as in the original mapping.
add_referred_subexpression(name, group, ...)	
add_subexpression(name, expr[, dimensions, ...])	Add a named subexpression.
create_clock_variables(clock[, prefix])	Convenience function to add the `t` and `dt` attributes of a `clock`.

Details

indices

    A dictionary given the index name for every array name

owner

    A reference to the *Group* owning these variables

add_arange (*name*, *size*, *start=0*, *dtype=<class 'numpy.int32'>*, *constant=True*, *read_only=True*, *unique=True*, *index=None*)

    Add an array, initialized with a range of integers.

> **Parameters** **name** : str
>
> > The name of the variable.
>
> **size** : int
>
> > The size of the array.
>
> **start** : int
>
> > The start value of the range.
>
> **dtype** : `dtype`, optional
>
> > The dtype used for storing the variable. If none is given, defaults to `np.int32`.
>
> **constant** : bool, optional
>
> > Whether the variable's value is constant during a run. Defaults to `True`.
>
> **read_only** : bool, optional
>
> > Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `True`.
>
> **index** : str, optional
>
> > The index to use for this variable. Defaults to `Variables.default_index`.
>
> **unique** : bool, optional
>
> > See *ArrayVariable*. Defaults to `True` here.

add_array (*name*, *size*, *dimensions=Dimension()*, *values=None*, *dtype=None*, *constant=False*, *read_only=False*, *scalar=False*, *unique=False*, *index=None*)

    Add an array (initialized with zeros).

> **Parameters** **name** : str
>
> > The name of the variable.
>
> **dimensions** : `Dimension`, optional
>
> > The physical dimensions of the variable.
>
> **size** : int
>
> > The size of the array.
>
> **values** : `ndarray`, optional
>
> > The values to initalize the array with. If not specified, the array is initialized to zero.
>
> **dtype** : `dtype`, optional

The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

constant : bool, optional

Whether the variable's value is constant during a run. Defaults to `False`.

scalar : bool, optional

Whether this is a scalar variable. Defaults to `False`, if set to `True`, also implies that `size()` equals 1.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

unique : bool, optional

See *ArrayVariable*. Defaults to `False`.

add_arrays(*names*, *size*, *dimensions=Dimension()*, *dtype=None*, *constant=False*, *read_only=False*, *scalar=False*, *unique=False*, *index=None*)

Adds several arrays (initialized with zeros) with the same attributes (size, units, etc.).

> **Parameters** **names** : list of str
>
> > The names of the variable.
> >
> > **dimensions** : `Dimension`, optional
> >
> > The physical dimensions of the variable.
> >
> > **size** : int
> >
> > The sizes of the arrays.
> >
> > **dtype** : `dtype`, optional
> >
> > The dtype used for storing the variables. If none is given, defaults to *core.default_float_dtype*.
> >
> > **constant** : bool, optional
> >
> > Whether the variables' values are constant during a run. Defaults to `False`.
> >
> > **scalar** : bool, optional
> >
> > Whether these are scalar variables. Defaults to `False`, if set to `True`, also implies that `size()` equals 1.
> >
> > **read_only** : bool, optional
> >
> > Whether these are read-only variables, i.e. variables that are set internally and cannot be changed by the user. Defaults to `False`.
> >
> > **index** : str, optional
> >
> > The index to use for these variables. Defaults to `Variables.default_index`.
> >
> > **unique** : bool, optional
> >
> > See *ArrayVariable*. Defaults to `False`.

add_auxiliary_variable(*name*, *dimensions=Dimension()*, *dtype=None*, *scalar=False*)
    Add an auxiliary variable (most likely one that is added automatically to abstract code, e.g. `_cond` for a threshold condition), specifying its type and unit for code generation.

        Parameters name : str

            The name of the variable

        dimensions : `Dimension`

            The physical dimensions of the variable.

        dtype : `dtype`, optional

            The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

        scalar : bool, optional

            Whether the variable is a scalar value (`True`) or vector-valued, e.g. defined for every neuron (`False`). Defaults to `False`.

add_constant(*name*, *value*, *dimensions=Dimension()*)
    Add a scalar constant (e.g. the number of neurons `N`).

        Parameters name : str

            The name of the variable

        value: reference to the variable value :

            The value of the constant.

        dimensions : `Dimension`, optional

            The physical dimensions of the variable. Note that the variable itself (as referenced by value) should never have units attached.

add_dynamic_array(*name*, *size*, *dimensions=Dimension()*, *values=None*, *dtype=None*, *constant=False*, *needs_reference_update=False*, *resize_along_first=False*, *read_only=False*, *unique=False*, *scalar=False*, *index=None*)
    Add a dynamic array.

        Parameters name : str

            The name of the variable.

        dimensions : `Dimension`, optional

             The physical dimensions of the variable.

        size : int or tuple of int

            The (initital) size of the array.

        values : `ndarray`, optional

            The values to initalize the array with. If not specified, the array is initialized to zero.

        dtype : `dtype`, optional

            The dtype used for storing the variable. If none is given, defaults to *core.default_float_dtype*.

        constant : bool, optional

            Whether the variable's value is constant during a run. Defaults to `False`.

needs_reference_update : bool, optional

Whether the code objects need a new reference to the underlying data at every time step. This should be set if the size of the array can be changed by other code objects. Defaults to `False`.

scalar : bool, optional

Whether this is a scalar variable. Defaults to `False`, if set to `True`, also implies that `size()` equals 1.

read_only : bool, optional

Whether this is a read-only variable, i.e. a variable that is set internally and cannot be changed by the user. Defaults to `False`.

index : str, optional

The index to use for this variable. Defaults to `Variables.default_index`.

unique : bool, optional

See *DynamicArrayVariable*. Defaults to `False`.

add_object(*name*, *obj*)

Add an arbitrary Python object. This is only meant for internal use and therefore only names starting with an underscore are allowed.

> **Parameters** **name** : str
>
> > The name used for this object (has to start with an underscore).
>
> **obj** : object
>
> > An arbitrary Python object that needs to be accessed directly from a `CodeObject`.

add_reference(*name*, *group*, *varname=None*, *index=None*)

Add a reference to a variable defined somewhere else (possibly under a different name). This is for example used in *Subgroup* and *Synapses* to refer to variables in the respective *NeuronGroup*.

> **Parameters** **name** : str
>
> > The name of the variable (in this group, possibly a different name from `var.name`).
>
> **group** : *Group*
>
> > The group from which `var()` is referenced
>
> **varname** : str, optional
>
> > The variable to refer to. If not given, defaults to `name`.
>
> **index** : str, optional
>
> > The index that should be used for this variable (defaults to `Variables. default_index`).

add_references(*group*, *varnames*, *index=None*)

Add all *Variable* objects from a name to *Variable* mapping with the same name as in the original mapping.

> **Parameters** **group** : *Group*
>
> > The group from which the `variables` are referenced
>
> **varnames** : iterable of str
>
> > The variables that should be referred to in the current group

>> **index** : str, optional

>>> The index to use for all the variables (defaults to `Variables.default_index`)

> **add_referred_subexpression**(*name*, *group*, *subexpr*, *index*)

> **add_subexpression**(*name*, *expr*, *dimensions=Dimension()*, *dtype=None*, *scalar=False*, *index=None*)
>> Add a named subexpression.

>>> **Parameters name** : str

>>>> The name of the subexpression.

>>> **dimensions** : `Dimension`

>>>> The physical dimensions of the subexpression.

>>> **expr** : str

>>>> The subexpression itself.

>>> **dtype** : `dtype`, optional

>>>> The dtype used for the expression. Defaults to *core.default_float_dtype*.

>>> **scalar** : bool, optional

>>>> Whether this is an expression only referring to scalar variables. Defaults to `False`

>>> **index** : str, optional

>>>> The index to use for this variable. Defaults to `Variables.default_index`.

> **create_clock_variables**(*clock*, *prefix=''*)
>> Convenience function to add the `t` and `dt` attributes of a `clock`.

>>> **Parameters clock** : *Clock*

>>>> The clock that should be used for `t` and `dt`.

>>> **prefix** : str, optional

>>>> A prefix for the variable names. Used for example in monitors to not confuse the dynamic array of recorded times with the current time in the recorded group.

Functions

get_dtype(obj)	Helper function to return the `numpy.dtype` of an arbitrary object.

get_dtype function

(*Shortest import*: `from brian2.core.variables import get_dtype`)

brian2.core.variables.**get_dtype**(*obj*)
> Helper function to return the `numpy.dtype` of an arbitrary object.

>> **Parameters obj** : object

>>> Any object (but typically some kind of number or array).

>> **Returns dtype** : `numpy.dtype`

>>> The type of the given object.

get_dtype_str(val)	Returns canonical string representation of the dtype of a value or dtype

get_dtype_str function

(*Shortest import*: `from brian2.core.variables import get_dtype_str`)

brian2.core.variables.**get_dtype_str**(*val*)

> Returns canonical string representation of the dtype of a value or dtype

> > **Returns dtype_str** : str

> > > The numpy dtype name

linked_var(group_or_variable[, name, index])	Represents a link target for setting a linked variable.

linked_var function

(*Shortest import*: `from brian2 import linked_var`)

brian2.core.variables.**linked_var**(*group_or_variable*, *name=None*, *index=None*)

> Represents a link target for setting a linked variable.

> > **Parameters group_or_variable** : *NeuronGroup* or *VariableView*

> > > Either a reference to the target *NeuronGroup* (e.g. G) or a direct reference to a *VariableView* object (e.g. `G.v`). In case only the group is specified, `name` has to be specified as well.

> > **name** : str, optional

> > > The name of the target variable, necessary if `group_or_variable` is a *NeuronGroup*.

> > **index** : str or *ndarray*, optional

> > > An indexing array (or the name of a state variable), providing a mapping from the entries in the link source to the link target.

Examples

```
>>> from brian2 import *
>>> G1 = NeuronGroup(10, 'dv/dt = -v / (10*ms) : volt')
>>> G2 = NeuronGroup(10, 'v : volt (linked)')
>>> G2.v = linked_var(G1, 'v')
>>> G2.v = linked_var(G1.v)  # equivalent
```

variables_by_owner(variables, owner)	

variables_by_owner function

(*Shortest import*: `from brian2.core.variables import variables_by_owner`)

brian2.core.variables.**variables_by_owner**(*variables*, *owner*)

6.7.3 devices package

Package providing the "devices" infrastructure.

device module

Module containing the *Device* base class as well as the *RuntimeDevice* implementation and some helper functions to access/set devices.

Exported members: *Device*, *RuntimeDevice*, *get_device()*, *set_device()*, `all_devices`, *reinit_devices*, *reinit_and_delete*, *reset_device*, *device*, *seed()*

Classes

CurrentDeviceProxy()	Method proxy for access to the currently active device

CurrentDeviceProxy class

(*Shortest import*: `from brian2.devices.device import CurrentDeviceProxy`)

class brian2.devices.device.**CurrentDeviceProxy**

    Bases: `object`

    Method proxy for access to the currently active device

Device()	Base Device object.

Device class

(*Shortest import*: `from brian2.devices import Device`)

class brian2.devices.device.**Device**

    Bases: `object`

    Base Device object.

Attributes

network_schedule	The network schedule that this device supports.

Methods

activate([build_on_run])	Called when this device is set as the current device.
add_array(var)	Add an array to this device.
build(**kwds)	For standalone projects, called when the project is ready to be built.
code_object(owner, name, abstract_code, ...)	
code_object_class([codeobj_class, fallback_pref])	Return `CodeObject` class according to input/default settings
delete([data, code, directory, force])	Delete code and/or data generated/stored by the device.
fill_with_array(var, arr)	Fill an array with the values given in another array.
get_array_name(var[, access_data])	Return a globally unique name for `var()`.
get_len(array)	Return the length of the array.
get_random_state()	Return a (pickable) representation of the current random number generator state.
init_with_arange(var, start, dtype)	Initialize an array with an integer range.
init_with_zeros(var, dtype)	Initialize an array with zeros.
insert_code(slot, code)	Insert code directly into a given slot in the device.
insert_device_code(slot, code)	
reinit()	Reinitialize the device.
resize(var, new_size)	Resize a `DynamicArrayVariable`.
resize_along_first(var, new_size)	
seed([seed])	Set the seed for the random number generator.
set_random_state(state)	Reset the random number generator state to a previously stored state (see *Device.get_random_state*).
spike_queue(source_start, source_end)	Create and return a new `SpikeQueue` for this *Device*.

Details

network_schedule
> The network schedule that this device supports. If the device only supports a specific, fixed schedule, it has to set this attribute to the respective schedule (see *Network.schedule* for details). If it supports arbitrary schedules, it should be set to `None` (the default).

activate(*build_on_run=True*, ***kwargs*)
> Called when this device is set as the current device.

add_array(*var*)
> Add an array to this device.
>
> > **Parameters var** : `ArrayVariable`

The array to add.

build(***kwds*)

For standalone projects, called when the project is ready to be built. Does nothing for runtime mode.

code_object(*owner*, *name*, *abstract_code*, *variables*, *template_name*, *variable_indices*, *codeobj_class=None*, *template_kwds=None*, *override_conditional_write=None*, *compiler_kwds=None*)

code_object_class(*codeobj_class=None*, *fallback_pref='codegen.target'*)

Return `CodeObject` class according to input/default settings

> **Parameters code_obj_class** : a `CodeObject` class, optional
>
> > If this is keyword is set to None or no arguments are given, this method will return the default.
>
> **fallback_pref** : str, optional
>
> > String describing which attribute of prefs to access to retrieve the 'default' target. Usually this is codegen.target, but in some cases we want to use object-specific targets such as codegen.string_expression_target.
>
> **Returns code_obj_class** : class
>
> > The `CodeObject` class that should be used

delete(*data=True*, *code=True*, *directory=True*, *force=False*)

Delete code and/or data generated/stored by the device.

> **Parameters data** : bool, optional
>
> > Whether to delete the data generated by the simulation (final values of state variables, data stored in monitors, etc.). Defaults to `True`.
>
> **code** : bool, optional
>
> > Whether to delete the code generated by the simulation. Includes the numerical values used for initialization of state variables in assignments not using strings. Defaults to `True`.
>
> **directory** : bool, optional
>
> > Whether to delete the project directory generated by the simulation. Will not delete directories that contain files not created by Brian unless the `force` option is specfied. Defaults to `True`.
>
> **force** : bool, optional
>
> > Whether to delete the project directory with all its content, even if it contains files that were not created by Brian. Useful only when the `directory` option is set to `True` as well. Defaults to `False`.

fill_with_array(*var*, *arr*)

Fill an array with the values given in another array.

> **Parameters var** : `ArrayVariable`
>
> > The array to fill.
>
> **arr** : ndarray
>
> > The array values that should be copied to `var()`.

get_array_name(*var*, *access_data=True*)

Return a globally unique name for `var()`.

> **Parameters access_data** : bool, optional

> For `DynamicArrayVariable` objects, specifying `True` here means the name for the underlying data is returned. If specifying `False`, the name of object itself is returned (e.g. to allow resizing).

> > **Returns name** : str

> > > The name for `var()`.

get_len(*array*)

Return the length of the array.

> **Parameters array** : `ArrayVariable`

> > The array for which the length is requested.

> **Returns l** : int

> > The length of the array.

get_random_state()

Return a (pickable) representation of the current random number generator state. Providing the returned object (e.g. a dict) to *`Device.set_random_state`* should restore the random number generator state.

> **Returns state** :

> > The state of the random number generator in a representation that can be passed as an argument to *`Device.set_random_state`*.

init_with_arange(*var*, *start*, *dtype*)

Initialize an array with an integer range.

> **Parameters var** : `ArrayVariable`

> > The array to fill with the integer range.

> **start** : int

> > The start value for the integer range

> **dtype** : `dtype`

> > The data type to use for the array.

init_with_zeros(*var*, *dtype*)

Initialize an array with zeros.

> **Parameters var** : `ArrayVariable`

> > The array to initialize with zeros.

> **dtype** : `dtype`

> > The data type to use for the array.

insert_code(*slot*, *code*)

Insert code directly into a given slot in the device. By default does nothing.

insert_device_code(*slot*, *code*)

reinit()

Reinitialize the device. For standalone devices, clears all the internal state of the device.

resize(*var*, *new_size*)

Resize a `DynamicArrayVariable`.

> **Parameters var** : `DynamicArrayVariable`

> > The variable that should be resized.

new_size : int

The new size of the variable

resize_along_first(*var*, *new_size*)

seed(*seed=None*)
Set the seed for the random number generator.

Parameters seed : int, optional

The seed value for the random number generator, or None (the default) to set a random
seed.

set_random_state(*state*)
Reset the random number generator state to a previously stored state (see *Device.get_random_state*).

Parameters state :

A random number generator state as provided by *Device.get_random_state*.

spike_queue(*source_start*, *source_end*)
Create and return a new SpikeQueue for this *Device*.

Parameters source_start : int

The start index of the source group (necessary for subgroups)

source_end : int

The end index of the source group (necessary for subgroups)

Dummy()	Dummy object

Dummy class

(*Shortest import*: from brian2.devices.device import Dummy)

class brian2.devices.device.**Dummy**
Bases: object

Dummy object

Methods

__call__(*args, **kwds)	Call self as a function.

Details

__call__(**args*, ***kwds*)
   Call self as a function.

Tutorials and examples using this

- Example *advanced/exprel_function*

- Example *frompapers/Stimberg_et_al_2018/example_2_gchi_astrocyte*

RuntimeDevice()	The default device used in Brian, state variables are stored as numpy arrays in memory.

RuntimeDevice class

(*Shortest import*: `from brian2.devices import RuntimeDevice`)

class brian2.devices.device.**RuntimeDevice**
   Bases: *brian2.devices.device.Device*

   The default device used in Brian, state variables are stored as numpy arrays in memory.

Attributes

arrays	Mapping from `Variable` objects to numpy arrays (or `DynamicArray` objects).

Methods

add_array(var)	Add an array to this device.
fill_with_array(var, arr)	Fill an array with the values given in another array.
get_array_name(var[, access_data])	Return a globally unique name for `var()`.
get_random_state()	Return a (pickable) representation of the current random number generator state.
get_value(var[, access_data])	
init_with_arange(var, start, dtype)	Initialize an array with an integer range.
init_with_zeros(var, dtype)	Initialize an array with zeros.
resize(var, new_size)	Resize a `DynamicArrayVariable`.
resize_along_first(var, new_size)	
seed([seed])	Set the seed for the random number generator.
set_random_state(state)	Reset the random number generator state to a previously stored state (see *Device.get_random_state*).
set_value(var, value)	

continues on next page

Table 217 – continued from previous page

spike_queue(source_start, source_end)	Create and return a new SpikeQueue for this *Device*.

Details

arrays
> Mapping from Variable objects to numpy arrays (or DynamicArray objects). Arrays in this dictionary will disappear as soon as the last reference to the Variable object used as a key is gone

add_array(*var*)
> Add an array to this device.
>
> > **Parameters var** : ArrayVariable
> >
> > > The array to add.

fill_with_array(*var*, *arr*)
> Fill an array with the values given in another array.
>
> > **Parameters var** : ArrayVariable
> >
> > > The array to fill.
> >
> > **arr** : ndarray
> >
> > > The array values that should be copied to var().

get_array_name(*var*, *access_data=True*)
> Return a globally unique name for var().
>
> > **Parameters access_data** : bool, optional
> >
> > > For DynamicArrayVariable objects, specifying True here means the name for the underlying data is returned. If specifying False, the name of object itself is returned (e.g. to allow resizing).
> >
> > **Returns name** : str
> >
> > > The name for var().

get_random_state()
> Return a (pickable) representation of the current random number generator state. Providing the returned object (e.g. a dict) to *Device.set_random_state* should restore the random number generator state.
>
> > **Returns state** :
> >
> > > The state of the random number generator in a representation that can be passed as an argument to *Device.set_random_state*.

get_value(*var*, *access_data=True*)

init_with_arange(*var*, *start*, *dtype*)
> Initialize an array with an integer range.
>
> > **Parameters var** : ArrayVariable
> >
> > > The array to fill with the integer range.
> >
> > **start** : int
> >
> > > The start value for the integer range
> >
> > **dtype** : dtype

The data type to use for the array.

init_with_zeros(*var*, *dtype*)
Initialize an array with zeros.

> **Parameters var** : `ArrayVariable`
>
>> The array to initialize with zeros.
>
> **dtype** : `dtype`
>
>> The data type to use for the array.

resize(*var*, *new_size*)
Resize a `DynamicArrayVariable`.

> **Parameters var** : `DynamicArrayVariable`
>
>> The variable that should be resized.
>
> **new_size** : int
>
>> The new size of the variable

resize_along_first(*var*, *new_size*)

seed(*seed=None*)
Set the seed for the random number generator.

> **Parameters seed** : int, optional
>
>> The seed value for the random number generator, or `None` (the default) to set a random seed.

set_random_state(*state*)
Reset the random number generator state to a previously stored state (see *Device.get_random_state*).

> **Parameters state** :
>
>> A random number generator state as provided by *Device.get_random_state*.

set_value(*var*, *value*)

spike_queue(*source_start*, *source_end*)
Create and return a new `SpikeQueue` for this *Device*.

> **Parameters source_start** : int
>
>> The start index of the source group (necessary for subgroups)
>
> **source_end** : int
>
>> The end index of the source group (necessary for subgroups)

Tutorials and examples using this

- Example *frompapers/Kremer_et_al_2011_barrel_cortex*

Functions

auto_target()	Automatically chose a code generation target (invoked when the *codegen.target* preference is set to `'auto'`.

auto_target function

(*Shortest import*: `from brian2.devices.device import auto_target`)

brian2.devices.device.**auto_target**()
> Automatically chose a code generation target (invoked when the *codegen.target* preference is set to `'auto'`. Caches its result so it only does the check once. Prefers cython > numpy.
>
> > **Returns target** : class derived from `CodeObject`
> >
> > > The target to use

get_device()	Gets the actve *Device* object

get_device function

(*Shortest import*: `from brian2 import get_device`)

brian2.devices.device.**get_device**()
> Gets the actve *Device* object

reinit_and_delete()	Calls *reinit_devices* and additionally deletes the files left behind by the standalone mode in the temporary directory.

reinit_and_delete function

(*Shortest import*: `from brian2.devices import reinit_and_delete`)

brian2.devices.device.**reinit_and_delete**()
> Calls *reinit_devices* and additionally deletes the files left behind by the standalone mode in the temporary directory. Silently suppresses errors that occur while deleting the directory.

reinit_devices()	Reinitialize all devices, call *Device.activate* again on the current device and reset the preferences.

reinit_devices function

(*Shortest import*: `from brian2.devices import reinit_devices`)

brian2.devices.device.**reinit_devices**()
> Reinitialize all devices, call *Device.activate* again on the current device and reset the preferences. Used as a "teardown" function in testing, if users want to reset their device (e.g. for multiple standalone runs in a single script), calling `device.reinit()` followed by `device.activate()` should normally be sufficient.

Notes

This also resets the *defaultclock*, i.e. a non-standard dt has to be set again.

reset_device([device])	Reset to a previously used device.

reset_device function

(*Shortest import*: from brian2.devices import reset_device)

brian2.devices.device.**reset_device**(*device=None*)

   Reset to a previously used device. Restores also the previously specified build options (see *set_device()*) for the device. Mostly useful for internal Brian code and testing on various devices.

   Parameters **device** : *Device* or str, optional

   The device to go back to. If none is specified, go back to the device chosen with *set_device()* before the current one.

seed([seed])	Set the seed for the random number generator.

seed function

(*Shortest import*: from brian2 import seed)

brian2.devices.device.**seed**(*seed=None*)

   Set the seed for the random number generator.

   Parameters **seed** : int, optional

   The seed value for the random number generator, or None (the default) to set a random seed.

Notes

This function delegates the call to *Device.seed* of the current device.

set_device(device[, build_on_run])	Set the device used for simulations.

set_device function

(*Shortest import*: from brian2 import set_device)

brian2.devices.device.**set_device**(*device*, *build_on_run=True*, ***kwargs*)

   Set the device used for simulations.

   Parameters **device** : *Device* or str

   The *Device* object or the name of the device.

   build_on_run : bool, optional

   Whether a call to *run()* (or *Network.run*) should directly trigger a *Device.build*. This is only relevant for standalone devices and means that a run call directly

triggers the start of a simulation. If the simulation consists of multiple run calls, set
build_on_run to False and call *Device.build* explicitly. Defaults to True.

> **kwargs** : dict, optional
>
>> Only relevant when build_on_run is True: additional arguments that will be given
>> to the *Device.build* call.

Objects

active_device	The currently active device (set with *set_device()*)

active_device object

(*Shortest import*: from brian2.devices.device import active_device)

brian2.devices.device.**active_device = <brian2.devices.device.RuntimeDevice
object>**

> The currently active device (set with *set_device()*)

device	Proxy object to access methods of the current device

device object

(*Shortest import*: from brian2 import device)

brian2.devices.device.**device = <brian2.devices.device.CurrentDeviceProxy
object>**

> Proxy object to access methods of the current device

runtime_device	The default device used in Brian, state variables are stored as numpy arrays in memory.

runtime_device object

(*Shortest import*: from brian2.devices.device import runtime_device)

brian2.devices.device.**runtime_device = <brian2.devices.device.RuntimeDevice
object>**

> The default device used in Brian, state variables are stored as numpy arrays in memory.

Subpackages

cpp_standalone package

Package implementing the C++ "standalone" Device and CodeObject.

`GSLcodeobject` **module**

Module containing CPPStandalone CodeObject for code generation for integration using the ODE solver provided in the GNU Scientific Library

Classes

GSLCPPStandaloneCodeObject(*args, **kw)	

GSLCPPStandaloneCodeObject class

(*Shortest import*: `from brian2.devices.cpp_standalone import GSLCPPStandaloneCodeObject`)

class `brian2.devices.cpp_standalone.GSLcodeobject.`**GSLCPPStandaloneCodeObject**(*args*, **kw*)

    Bases: *brian2.codegen.codeobject.CodeObject*

`codeobject` **module**

Module implementing the C++ "standalone" `CodeObject`

Exported members: *CPPStandaloneCodeObject*

Classes

CPPStandaloneCodeObject(*args, **kw)	C++ standalone code object

CPPStandaloneCodeObject class

(*Shortest import*: `from brian2.devices.cpp_standalone import CPPStandaloneCodeObject`)

class `brian2.devices.cpp_standalone.codeobject.`**CPPStandaloneCodeObject**(*args*, **kw*)

    Bases: *brian2.codegen.codeobject.CodeObject*

    C++ standalone code object

    The `code` should be a *MultiTemplate* object with two macros defined, `main` (for the main loop code) and `support_code` for any support code (e.g. function definitions).

    #### Attributes

before_after_blocks	Store whether this code object defines before/after blocks

Methods

[`__call__`](#)(**kwds)	Call self as a function.
[`compile_block`](#)(block)	
[`run_block`](#)(block)	

Details

before_after_blocks
> Store whether this code object defines before/after blocks

__call__(***kwds*)
> Call self as a function.

compile_block(*block*)

run_block(*block*)

Functions

[`generate_rand_code`](#)(rand_func, owner)

generate_rand_code function

(*Shortest import*: from brian2.devices.cpp_standalone.codeobject import generate_rand_code)

brian2.devices.cpp_standalone.codeobject.**generate_rand_code**(*rand_func*, *owner*)

[`openmp_pragma`](#)(pragma_type)

openmp_pragma function

(*Shortest import*: from brian2.devices.cpp_standalone.codeobject import openmp_pragma)

brian2.devices.cpp_standalone.codeobject.**openmp_pragma**(*pragma_type*)

`device` **module**

Module implementing the C++ "standalone" device.

Classes

CPPStandaloneDevice()	The `Device` used for C++ standalone simulations.

CPPStandaloneDevice class

(*Shortest import*: `from brian2.devices.cpp_standalone.device import CPPStandaloneDevice`)

class brian2.devices.cpp_standalone.device.**CPPStandaloneDevice**
    Bases: *brian2.devices.device.Device*

    The `Device` used for C++ standalone simulations.

Attributes

arange_arrays	List of all arrays to be filled with numbers (list of (var, varname, start) tuples
array_cache	Dictionary mapping `ArrayVariable` objects to their value or to `None` if the value (potentially) depends on executed code.
arrays	Dictionary mapping `ArrayVariable` objects to their globally unique name
build_on_run	Whether a run should trigger a build
build_options	build options
code_lines	Code lines that have been manually added with `device.insert_code()` Dictionary mapping slot names to lists of lines.
dynamic_arrays	List of all dynamic arrays Dictionary mapping `DynamicArrayVariable` objects with 1 dimension to their globally unique name
dynamic_arrays_2d	Dictionary mapping `DynamicArrayVariable` objects with 2 dimensions to their globally unique name
enable_profiling	Whether to generate profiling information (stored in an instance variable to be accessible during CodeObject generation)
has_been_run	Whether the simulation has been run
profiled_codeobjects	CodeObjects that use profiling (users can potentially enable profiling only for a subset of runs)
project_dir	The directory which contains the generated code and results
static_arrays	Dict of all static saved arrays
synapses	Set of all existing synapses
zero_arrays	List of all arrays to be filled with zeros (list of (var, varname))

Methods

[add_array](var)	Add an array to this device.
[build]([directory, compile, run, debug, ...])	Build the project
[check_openmp_compatible](nb_threads)	
[code_object](owner, name, abstract_code, ...)	
[code_object_class]([codeobj_class, fall-back_pref])	Return `CodeObject` class (either `CPPStandaloneCodeObject` class or input)
[compile_source](directory, compiler, debug, clean)	
[copy_source_files](writer, directory)	
[delete]([code, data, directory, force])	Delete code and/or data generated/stored by the device.
[fill_with_array](var, arr)	Fill an array with the values given in another array.
[freeze](code, ns)	
[generate_codeobj_source](writer)	
[generate_main_source](writer)	
[generate_makefile](writer, compiler, ...)	
[generate_network_source](writer, compiler)	
[generate_objects_source](writer, ...)	
[generate_run_source](writer)	
[generate_synapses_classes_source](writer)	
[get_array_filename](var[, basedir])	Return a file name for a variable.
[get_array_name](var[, access_data])	Return a globally unique name for `var()`.
[get_value](var[, access_data])	
[init_with_arange](var, start, dtype)	Initialize an array with an integer range.
[init_with_zeros](var, dtype)	Initialize an array with zeros.
[insert_code](slot, code)	Insert code directly into main.cpp
[network_get_profiling_info](net)	
[network_restore](net, *args, **kwds)	
[network_run](net, duration[, report, ...])	
[network_store](net, *args, **kwds)	
[reinit]()	Reinitialize the device.
[resize](var, new_size)	Resize a `DynamicArrayVariable`.

Table 236 – continued from previous page

run(directory, with_output, run_args)	
run_function(name[, include_in_parent])	Context manager to divert code into a function
seed([seed])	Set the seed for the random number generator.
spike_queue(source_start, source_end)	Create and return a new `SpikeQueue` for this `Device`.
static_array(name, arr)	
variableview_get_subexpression_with_index_array(...)	
variableview_get_with_expression(...[, ...])	
variableview_set_with_index_array(...)	
write_static_arrays(directory)	

Details

arange_arrays

    List of all arrays to be filled with numbers (list of (var, varname, start) tuples

array_cache

    Dictionary mapping `ArrayVariable` objects to their value or to `None` if the value (potentially) depends on executed code. This mechanism allows to access state variables in standalone mode if their value is known at run time

arrays

    Dictionary mapping `ArrayVariable` objects to their globally unique name

build_on_run

    Whether a run should trigger a build

build_options

    build options

code_lines

    Code lines that have been manually added with `device.insert_code()` Dictionary mapping slot names to lists of lines. Note that the main slot is handled separately as part of `main_queue`

dynamic_arrays

    List of all dynamic arrays Dictionary mapping `DynamicArrayVariable` objects with 1 dimension to their globally unique name

dynamic_arrays_2d

    Dictionary mapping `DynamicArrayVariable` objects with 2 dimensions to their globally unique name

enable_profiling

    Whether to generate profiling information (stored in an instance variable to be accessible during CodeObject generation)

has_been_run

    Whether the simulation has been run

profiled_codeobjects

    CodeObjects that use profiling (users can potentially enable profiling only for a subset of runs)

project_dir
> The directory which contains the generated code and results

static_arrays
> Dict of all static saved arrays

synapses
> Set of all existing synapses

zero_arrays
> List of all arrays to be filled with zeros (list of (var, varname))

add_array(*var*)
> Add an array to this device.

>> **Parameters var** : `ArrayVariable`

>>> The array to add.

build(*directory='output'*, *compile=True*, *run=True*, *debug=False*, *clean=False*, *with_output=True*,
> *additional_source_files=None*, *run_args=None*, *direct_call=True*, ***kwds*)
> Build the project

> TODO: more details

>> **Parameters directory** : str, optional

>>> The output directory to write the project to, any existing files will be overwritten. If the
given directory name is `None`, then a temporary directory will be used (used in the test
suite to avoid problems when running several tests in parallel). Defaults to `'output'`.

>>> **compile** : bool, optional

>>> Whether or not to attempt to compile the project. Defaults to `True`.

>>> **run** : bool, optional

>>> Whether or not to attempt to run the built project if it successfully builds. Defaults to
`True`.

>>> **debug** : bool, optional

>>> Whether to compile in debug mode. Defaults to `False`.

>>> **with_output** : bool, optional

>>> Whether or not to show the `stdout` of the built program when run. Output will be shown
in case of compilation or runtime error. Defaults to `True`.

>>> **clean** : bool, optional

>>> Whether or not to clean the project before building. Defaults to `False`.

>>> **additional_source_files** : list of str, optional

>>> A list of additional `.cpp` files to include in the build.

>>> **direct_call** : bool, optional

>>> Whether this function was called directly. Is used internally to distinguish an automatic
build due to the `build_on_run` option from a manual `device.build` call.

check_openmp_compatible(*nb_threads*)

code_object(*owner*, *name*, *abstract_code*, *variables*, *template_name*, *variable_indices*, *codeobj_class=None*,
> *template_kwds=None*, *override_conditional_write=None*, *compiler_kwds=None*)

code_object_class(*codeobj_class=None*, *fallback_pref=None*)
Return `CodeObject` class (either `CPPStandaloneCodeObject` class or input)

> **Parameters codeobj_class** : a `CodeObject` class, optional
>
> > If this is keyword is set to None or no arguments are given, this method will return the default (`CPPStandaloneCodeObject` class).
>
> **fallback_pref** : str, optional
>
> > For the cpp_standalone device this option is ignored.
>
> **Returns codeobj_class** : class
>
> > The `CodeObject` class that should be used

compile_source(*directory*, *compiler*, *debug*, *clean*)

copy_source_files(*writer*, *directory*)

delete(*code=True*, *data=True*, *directory=True*, *force=False*)
Delete code and/or data generated/stored by the device.

> **Parameters data** : bool, optional
>
> > Whether to delete the data generated by the simulation (final values of state variables, data stored in monitors, etc.). Defaults to `True`.
>
> **code** : bool, optional
>
> > Whether to delete the code generated by the simulation. Includes the numerical values used for initialization of state variables in assignments not using strings. Defaults to `True`.
>
> **directory** : bool, optional
>
> > Whether to delete the project directory generated by the simulation. Will not delete directories that contain files not created by Brian unless the `force` option is specfied. Defaults to `True`.
>
> **force** : bool, optional
>
> > Whether to delete the project directory with all its content, even if it contains files that were not created by Brian. Useful only when the `directory` option is set to `True` as well. Defaults to `False`.

fill_with_array(*var*, *arr*)
Fill an array with the values given in another array.

> **Parameters var** : `ArrayVariable`
>
> > The array to fill.
>
> **arr** : [ndarray](#)
>
> > The array values that should be copied to `var()`.

freeze(*code*, *ns*)

generate_codeobj_source(*writer*)

generate_main_source(*writer*)

generate_makefile(*writer*, *compiler*, *compiler_flags*, *linker_flags*, *nb_threads*, *debug*)

generate_network_source(*writer*, *compiler*)

generate_objects_source(*writer*, *arange_arrays*, *synapses*, *static_array_specs*, *networks*)

generate_run_source(*writer*)

generate_synapses_classes_source(*writer*)

get_array_filename(*var*, *basedir='results'*)
    Return a file name for a variable.

> **Parameters var** : `ArrayVariable`
>
> > The variable to get a filename for.
>
> **basedir** : str
>
> > The base directory for the filename, defaults to `'results'`.
>
> **Returns** :
>
> ——- :
>
> **filename** : str
>
> > A filename of the form `'results/'+varname+'_'+str(hash(varname))`,
> > where varname is the name returned by *get_array_name*.

> ### Notes
>
> The reason that the filename is not simply `'results/' + varname` is that this could lead to file names
> that are not unique in file systems that are not case sensitive (e.g. on Windows).

get_array_name(*var*, *access_data=True*)
    Return a globally unique name for `var()`.

> **Parameters access_data** : bool, optional
>
> > For `DynamicArrayVariable` objects, specifying `True` here means the name for the
> > underlying data is returned. If specifying `False`, the name of object itself is returned
> > (e.g. to allow resizing).

get_value(*var*, *access_data=True*)

init_with_arange(*var*, *start*, *dtype*)
    Initialize an array with an integer range.

> **Parameters var** : `ArrayVariable`
>
> > The array to fill with the integer range.
>
> **start** : int
>
> > The start value for the integer range
>
> **dtype** : `dtype`
>
> > The data type to use for the array.

init_with_zeros(*var*, *dtype*)
    Initialize an array with zeros.

> **Parameters var** : `ArrayVariable`
>
> > The array to initialize with zeros.
>
> **dtype** : `dtype`
>
> > The data type to use for the array.

insert_code(*slot*, *code*)
  Insert code directly into main.cpp

network_get_profiling_info(*net*)

network_restore(*net*, **args*, ***kwds*)

network_run(*net*, *duration*, *report=None*, *report_period=10.0 * second*, *namespace=None*, *profile=None*,
    level=0, ***kwds*)

network_store(*net*, **args*, ***kwds*)

reinit()
  Reinitialize the device. For standalone devices, clears all the internal state of the device.

resize(*var*, *new_size*)
  Resize a `DynamicArrayVariable`.

>   **Parameters var** : `DynamicArrayVariable`
>
>   The variable that should be resized.
>
>   **new_size** : int
>
>   The new size of the variable

run(*directory*, *with_output*, *run_args*)

run_function(*name*, *include_in_parent=True*)
  Context manager to divert code into a function

  Code that happens within the scope of this context manager will go into the named function.

>   **Parameters name** : str
>
>   The name of the function to divert code into.
>
>   **include_in_parent** : bool
>
>   Whether or not to include a call to the newly defined function in the parent context.

seed(*seed=None*)
  Set the seed for the random number generator.

>   **Parameters seed** : int, optional
>
>   The seed value for the random number generator, or `None` (the default) to set a random
>   seed.

spike_queue(*source_start*, *source_end*)
  Create and return a new `SpikeQueue` for this `Device`.

>   **Parameters source_start** : int
>
>   The start index of the source group (necessary for subgroups)
>
>   **source_end** : int
>
>   The end index of the source group (necessary for subgroups)

static_array(*name*, *arr*)

variableview_get_subexpression_with_index_array(*variableview*, *item*,
    run_namespace=None)

variableview_get_with_expression(*variableview*, *code*, *run_namespace=None*)

variableview_set_with_index_array(*variableview*, *item*, *value*, *check_units*)

write_static_arrays(*directory*)

CPPWriter(project_dir)

Methods

CPPWriter class

(*Shortest import*: `from brian2.devices.cpp_standalone.device import CPPWriter`)

class `brian2.devices.cpp_standalone.device.`**CPPWriter**(*project_dir*)

    Bases: `object`

Methods

---

write(filename, contents)

Details

write(*filename*, *contents*)

RunFunctionContext(name, include_in_parent)

RunFunctionContext class

(*Shortest import*: `from brian2.devices.cpp_standalone.device import RunFunctionContext`)

class `brian2.devices.cpp_standalone.device.`**RunFunctionContext**(*name*,
include_in_parent)

    Bases: `object`

Functions

---

invert_dict(x)

---

invert_dict function

(*Shortest import*: `from brian2.devices.cpp_standalone.device import invert_dict`)

brian2.devices.cpp_standalone.device.**invert_dict**(*x*)

Objects

cpp_standalone_device	The `Device` used for C++ standalone simulations.

cpp_standalone_device object

(*Shortest import*: `from brian2.devices.cpp_standalone import cpp_standalone_device`)

brian2.devices.cpp_standalone.device.**cpp_standalone_device = <brian2.devices.cpp_standalone.device.CPPStandaloneDevice object>**

> The `Device` used for C++ standalone simulations.

6.7.4 equations package

Module handling equations and "code strings", expressions or statements, used for example for the reset and threshold definition of a neuron.

Exported members: *Equations*, *Expression*, *Statements*

codestrings module

Module defining *CodeString*, a class for a string of code together with information about its namespace. Only serves as a parent class, its subclasses *Expression* and *Statements* are the ones that are actually used.

Exported members: *Expression*, *Statements*

Classes

CodeString(code)	A class for representing "code strings", i.e. a single Python expression or a sequence of Python statements.

CodeString class

(*Shortest import*: `from brian2.equations.codestrings import CodeString`)

class brian2.equations.codestrings.**CodeString**(*code*)

> Bases: `collections.abc.Hashable`

> A class for representing "code strings", i.e. a single Python expression or a sequence of Python statements.

> > **Parameters code** : str

> > > The code string, may be an expression or a statement(s) (possibly multi-line).

Attributes

code	The code string

Details

code
> The code string

Expression([code, sympy_expression])	Class for representing an expression.

Expression class

(*Shortest import*: `from brian2 import Expression`)

class brian2.equations.codestrings.**Expression**(*code=None*, *sympy_expression=None*)
> Bases: *brian2.equations.codestrings.CodeString*

> Class for representing an expression.

> > **Parameters code** : str, optional

> > > The expression. Note that the expression has to be written in a form that is parseable by sympy. Alternatively, a sympy expression can be provided (in the `sympy_expression` argument).

> > **sympy_expression** : sympy expression, optional

> > > A sympy expression. Alternatively, a plain string expression can be provided (in the `code` argument).

Attributes

stochastic_variables	Stochastic variables in this expression

Methods

split_stochastic()	Split the expression into a stochastic and non-stochastic part.

Details

stochastic_variables

    Stochastic variables in this expression

split_stochastic()

    Split the expression into a stochastic and non-stochastic part.

    Splits the expression into a tuple of one *Expression* objects f (the non-stochastic part) and a dictionary mapping stochastic variables to *Expression* objects. For example, an expression of the form `f + g * xi_1 + h * xi_2` would be returned as: `(f, {'xi_1': g, 'xi_2': h})` Note that the *Expression* objects for the stochastic parts do not include the stochastic variable itself.

        Returns (f, d) : (*Expression*, dict)

            A tuple of an *Expression* object and a dictionary, the first expression being the non-stochastic part of the equation and the dictionary mapping stochastic variables (`xi` or starting with `xi_`) to *Expression* objects. If no stochastic variable is present in the code string, a tuple (`self, None`) will be returned with the unchanged *Expression* object.

Statements(code)	Class for representing statements.

Statements class

(*Shortest import*: `from brian2 import Statements`)

class brian2.equations.codestrings.**Statements**(*code*)

    Bases: *brian2.equations.codestrings.CodeString*

    Class for representing statements.

        Parameters code : str

            The statement or statements. Several statements can be given as a multi-line string or separated by semicolons.

Notes

Currently, the implementation of this class does not add anything to *CodeString*, but it should be used instead of that class for clarity and to allow for future functionality that is only relevant to statements and not to expressions.

Functions

is_constant_over_dt(expression, variables, ...)	Check whether an expression can be considered as constant over a time step.

is_constant_over_dt function

(*Shortest import*: `from brian2.equations.codestrings import is_constant_over_dt`)

`brian2.equations.codestrings.`**`is_constant_over_dt`**(*expression*, *variables*, *dt_value*)

Check whether an expression can be considered as constant over a time step. This is *not* the case when the expression either:

1. contains the variable `t` (except as the argument of a function that can be considered as constant over a time step, e.g. a *TimedArray* with a dt equal to or greater than the dt used to evaluate this expression)

2. refers to a stateful function such as `rand()`.

> **Parameters expression** : `sympy.Expr`
>
> > The (sympy) expression to analyze
>
> **variables** : dict
>
> > The variables dictionary.
>
> **dt_value** : float or None
>
> > The length of a timestep (without units), can be `None` if the time step is not yet known.
>
> **Returns is_constant** : bool
>
> > Whether the expression can be considered to be constant over a time step.

equations module

Differential equations for Brian models.

Exported members: Equations

Classes

EquationError	Exception type related to errors in an equation definition.

EquationError class

(*Shortest import*: `from brian2.equations.equations import EquationError`)

class `brian2.equations.equations.`**`EquationError`**

Bases: `Exception`

Exception type related to errors in an equation definition.

Equations(eqns, **kwds)	Container that stores equations from which models can be created.

Equations class

(*Shortest import*: `from brian2 import Equations`)

class `brian2.equations.equations.`**`Equations`**(*eqns*, ***kwds*)

    Bases: `collections.abc.Hashable`, `collections.abc.Mapping`

    Container that stores equations from which models can be created.

    String equations can be of any of the following forms:

        1. `dx/dt = f : unit (flags)` (differential equation)

        2. `x = f : unit (flags)` (equation)

        3. `x : unit (flags)` (parameter)

    String equations can span several lines and contain Python-style comments starting with #

        Parameters eqs : `str` or list of *`SingleEquation`* objects

            A multiline string of equations (see above) – for internal purposes also a list of *`SingleE-`*
            `quation` objects can be given. This is done for example when adding new equations to
            implement the refractory mechanism. Note that in this case the variable names are not
            checked to allow for "internal names", starting with an underscore.

        kwds: keyword arguments :

            Keyword arguments can be used to replace variables in the equation string. Arguments
            have to be of the form `varname=replacement`, where `varname` has to correspond
            to a variable name in the given equation. The replacement can be either a string (replacing
            a name with a new name, e.g. `tau='tau_e'`) or a value (replacing the variable name
            with the value, e.g. `tau=tau_e` or `tau=10*ms`).

Attributes

`_substituted_expressions`	Cache for equations with the subexpressions substituted
`dependencies`	Calculate the dependencies of all differential equations and subexpressions.
`diff_eq_expressions`	A list of (variable name, expression) tuples of all differential equations.
`diff_eq_names`	All differential equation names.
`dimensions`	Dictionary of all internal variables and their corresponding physical dimensions.
`eq_expressions`	A list of (variable name, expression) tuples of all equations.
`eq_names`	All equation names (including subexpressions).
`identifier_checks`	A set of functions that are used to check identifiers (class attribute).
`identifiers`	Set of all identifiers used in the equations, excluding the variables defined in the equations
`is_stochastic`	Whether the equations are stochastic.
`names`	All variable names defined in the equations.
`ordered`	A list of all equations, sorted according to the order in which they should be updated

<div align="right">continues on next page</div>

Table 251 – continued from previous page

parameter_names	All parameter names.
stochastic_type	Returns the type of stochastic differential equations (additivive or multiplicative).
stochastic_variables	
subexpr_names	All subexpression names.

Methods

check_flags(allowed_flags[, incompatible_flags])	Check the list of flags.
check_identifier(identifier)	Perform all the registered checks.
check_identifiers()	Check all identifiers for conformity with the rules.
check_units(group, run_namespace)	Check all the units for consistency.
get_substituted_expressions([variables, ...])	Return a list of (`varname`, `expr`) tuples, containing all differential equations (and optionally subexpressions) with all the subexpression variables substituted with the respective expressions.
register_identifier_check(func)	Register a function for checking identifiers.
substitute(**kwds)	

Details

_substituted_expressions
    Cache for equations with the subexpressions substituted

dependencies
    Calculate the dependencies of all differential equations and subexpressions.

diff_eq_expressions
    A list of (variable name, expression) tuples of all differential equations.

diff_eq_names
    All differential equation names.

dimensions
    Dictionary of all internal variables and their corresponding physical dimensions.

eq_expressions
    A list of (variable name, expression) tuples of all equations.

eq_names
    All equation names (including subexpressions).

identifier_checks
    A set of functions that are used to check identifiers (class attribute). Functions can be registered with the static method *Equations.register_identifier_check* and will be automatically used when checking identifiers

identifiers
    Set of all identifiers used in the equations, excluding the variables defined in the equations

is_stochastic
    Whether the equations are stochastic.

names
    All variable names defined in the equations.

ordered
    A list of all equations, sorted according to the order in which they should be updated

parameter_names
    All parameter names.

stochastic_type
    Returns the type of stochastic differential equations (additivive or multiplicative). The system is only classified as `additive` if *all* equations have only additive noise (or no noise).

>   **Returns type** : str
>
>>   Either `None` (no noise variables), `'additive'` (factors for all noise variables are independent of other state variables or time), `'multiplicative'` (at least one of the noise factors depends on other state variables and/or time).

stochastic_variables

subexpr_names
    All subexpression names.

check_flags (*allowed_flags*, *incompatible_flags=None*)
    Check the list of flags.

>   **Parameters allowed_flags** : dict
>
>>   A dictionary mapping equation types (PARAMETER, DIFFERENTIAL_EQUATION, SUBEXPRESSION) to a list of strings (the allowed flags for that equation type)
>
>   **incompatible_flags** : list of tuple
>
>   A list of flag combinations that are not allowed for the same equation.
>
>   **Notes** :
>
>   —— :
>
>   **Not specifying allowed flags for an equation type is the same as** :
>
>   **specifying an empty list for it.** :

>   **Raises**
>
>   **ValueError** If any flags are used that are not allowed.

static check_identifier (*identifier*)
    Perform all the registered checks. Checks can be registered via *Equations.register_identifier_check*.

>   **Parameters identifier** : str
>
>>   The identifier that should be checked

Raises

ValueError If any of the registered checks fails.

check_identifiers()
Check all identifiers for conformity with the rules.

Raises

ValueError If an identifier does not conform to the rules.

See also:

Equations.check_identifier The function that is called for each identifier.

check_units(*group*, *run_namespace*)
Check all the units for consistency.

> **Parameters group** : *Group*
>
> > The group providing the context
>
> **run_namespace** : dict-like, optional
>
> > An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).
>
> **level** : int, optional
>
> > How much further to go up in the stack to find the calling frame

Raises

DimensionMismatchError In case of any inconsistencies.

get_substituted_expressions(*variables=None*, *include_subexpressions=False*)
Return a list of (`varname`, `expr`) tuples, containing all differential equations (and optionally subexpressions) with all the subexpression variables substituted with the respective expressions.

> **Parameters variables** : dict, optional
>
> > A mapping of variable names to `Variable`/*Function* objects.
>
> **include_subexpressions** : bool
>
> > Whether also to return substituted subexpressions. Defaults to `False`.
>
> **Returns expr_tuples** : list of (str, `CodeString`)
>
> > A list of (`varname`, `expr`) tuples, where `expr` is a `CodeString` object with all subexpression variables substituted with the respective expression.

static register_identifier_check(*func*)
Register a function for checking identifiers.

> **Parameters func** : callable
>
> > The function has to receive a single argument, the name of the identifier to check, and raise a ValueError if the identifier violates any rule.

substitute(***kwds*)

Tutorials and examples using this

- Example *COBAHH*

- Example *IF_curve_Hodgkin_Huxley*

- Example *advanced/COBAHH_approximated*

- Example *advanced/float_32_64_benchmark*

- Example *frompapers/Clopath_et_al_2010_homeostasis*

- Example *frompapers/Clopath_et_al_2010_no_homeostasis*

- Example *frompapers/Destexhe_et_al_1998*

- Example *frompapers/Diesmann_et_al_1999*

- Example *frompapers/Hindmarsh_Rose_1984*

- Example *frompapers/Jansen_Rit_1995_single_column*

- Example *frompapers/Rossant_et_al_2011bis*

SingleEquation(type, varname, dimensions[, ...])	Class for internal use, encapsulates a single equation or parameter.

SingleEquation class

(*Shortest import*: `from brian2.equations.equations import SingleEquation`)

class brian2.equations.equations.**SingleEquation**(*type*, *varname*, *dimensions*, *var_type='float'*, *expr=None*, *flags=None*)

   Bases: `collections.abc.Hashable`, *brian2.utils.caching.CacheKey*

   Class for internal use, encapsulates a single equation or parameter.

Note: This class should never be used directly, it is only useful as part of the *Equations* class.

   Parameters type : {PARAMETER, DIFFERENTIAL_EQUATION, SUBEXPRESSION}

      The type of the equation.

   varname : str

      The variable that is defined by this equation.

   dimensions : `Dimension`

      The physical dimensions of the variable

   var_type : {FLOAT, INTEGER, BOOLEAN}

      The type of the variable (floating point value or boolean).

   expr : *Expression*, optional

      The expression defining the variable (or `None` for parameters).

flags: list of str, optional :

A list of flags that give additional information about this equation. What flags are possible depends on the type of the equation and the context.

Attributes

identifiers	All identifiers in the RHS of this equation.
stochastic_variables	Stochastic variables in the RHS of this equation
unit	The *Unit* of this equation.

Details

identifiers
> All identifiers in the RHS of this equation.

stochastic_variables
> Stochastic variables in the RHS of this equation

unit
> The *Unit* of this equation.

Functions

check_identifier_basic(identifier)	Check an identifier (usually resulting from an equation string provided by the user) for conformity with the rules.

check_identifier_basic function

(*Shortest import*: `from brian2.equations.equations import check_identifier_basic`)

brian2.equations.equations.**check_identifier_basic**(*identifier*)

> Check an identifier (usually resulting from an equation string provided by the user) for conformity with the rules. The rules are:

1. Only ASCII characters

2. Starts with a character, then mix of alphanumerical characters and underscore

3. Is not a reserved keyword of Python

> **Parameters identifier** : str
>
> > The identifier that should be checked

Raises

    SyntaxError If the identifier does not conform to the above rules.

[*check_identifier_constants*](identifier)	Make sure that identifier names do not clash with function names.

check_identifier_constants function

(*Shortest import*: `from brian2.equations.equations import check_identifier_constants`)

`brian2.equations.equations.`**`check_identifier_constants`**(*identifier*)
    Make sure that identifier names do not clash with function names.

[*check_identifier_functions*](identifier)	Make sure that identifier names do not clash with function names.

check_identifier_functions function

(*Shortest import*: `from brian2.equations.equations import check_identifier_functions`)

`brian2.equations.equations.`**`check_identifier_functions`**(*identifier*)
    Make sure that identifier names do not clash with function names.

[*check_identifier_reserved*](identifier)	Check that an identifier is not using a reserved special variable name.

check_identifier_reserved function

(*Shortest import*: `from brian2.equations.equations import check_identifier_reserved`)

`brian2.equations.equations.`**`check_identifier_reserved`**(*identifier*)
    Check that an identifier is not using a reserved special variable name. The special variables are: 't', 'dt', and 'xi', as well as everything starting with `xi_`.

        Parameters identifier: str :

            The identifier that should be checked

    Raises

    SyntaxError If the identifier is a special variable name.

[*check_identifier_units*](identifier)	Make sure that identifier names do not clash with unit names.

check_identifier_units function

(*Shortest import*: `from brian2.equations.equations import check_identifier_units`)

`brian2.equations.equations.`**`check_identifier_units`**(*identifier*)
    Make sure that identifier names do not clash with unit names.

check_subexpressions(group, equations, ...)	Checks the subexpressions in the equations and raises an error if a subexpression refers to stateful functions without being marked as "constant over dt".

check_subexpressions function

(*Shortest import*: `from brian2.equations.equations import check_subexpressions`)

`brian2.equations.equations.`**`check_subexpressions`**(*group*, *equations*, *run_namespace*)
    Checks the subexpressions in the equations and raises an error if a subexpression refers to stateful functions without being marked as "constant over dt".

> **Parameters group** : *Group*
>
> > The group providing the context.
>
> **equations** : *Equations*
>
> > The equations to check.
>
> **run_namespace** : dict
>
> > The run namespace for resolving variables.

Raises

> **SyntaxError** For subexpressions not marked as "constant over dt" that refer to stateful functions.

dimensions_and_type_from_string(unit_string)	Returns the physical dimensions that results from evaluating a string like "siemens / metre ** 2", allowing for the special string "1" to signify dimensionless units, the string "boolean" for a boolean and "integer" for an integer variable.

dimensions_and_type_from_string function

(*Shortest import*: `from brian2.equations.equations import dimensions_and_type_from_string`)

`brian2.equations.equations.`**`dimensions_and_type_from_string`**(*unit_string*)
    Returns the physical dimensions that results from evaluating a string like "siemens / metre ** 2", allowing for the special string "1" to signify dimensionless units, the string "boolean" for a boolean and "integer" for an integer variable.

> **Parameters unit_string** : str
>
> > The string that should evaluate to a unit

Returns d, type : (`Dimension`, {FLOAT, INTEGER or BOOL})

> The resulting physical dimensions and the type of the variable.

Raises

`ValueError` If the string cannot be evaluated to a unit.

---

extract_constant_subexpressions(eqs)

---

extract_constant_subexpressions function

(*Shortest import*: from brian2.equations.equations import extract_constant_subexpressions)

brian2.equations.equations.**extract_constant_subexpressions**(*eqs*)

---

is_stateful(expression, variables)	Whether the given expression refers to stateful functions (and is therefore not guaranteed to give the same result if called repetively).

---

is_stateful function

(*Shortest import*: from brian2.equations.equations import is_stateful)

brian2.equations.equations.**is_stateful**(*expression*, *variables*)

> Whether the given expression refers to stateful functions (and is therefore not guaranteed to give the same result if called repetively).

Parameters expression : `sympy.Expression`

> The sympy expression to check.

variables : dict

> The dictionary mapping variable names to `Variable` or *Function* objects.

Returns stateful : bool

> `True`, if the given expression refers to a stateful function like `rand()` and `False` otherwise.

---

parse_string_equations(eqns)	Parse a string defining equations.

---

parse_string_equations function

(*Shortest import*: `from brian2.equations.equations import parse_string_equations`)

brian2.equations.equations.**parse_string_equations**(*eqns*)

> Parse a string defining equations.

> > **Parameters eqns** : str

> > > The (possibly multi-line) string defining the equations. See the documentation of the *Equations* class for details.

> > **Returns equations** : dict

> > > A dictionary mapping variable names to *Equations* objects

refractory module

Module implementing Brian's refractory mechanism.

Exported members: *add_refractoriness*

Functions

add_refractoriness(eqs)	Extends a given set of equations with the refractory mechanism.

add_refractoriness function

(*Shortest import*: `from brian2.equations.refractory import add_refractoriness`)

brian2.equations.refractory.**add_refractoriness**(*eqs*)

> Extends a given set of equations with the refractory mechanism. New parameters are added and differential equations with the "unless refractory" flag are changed so that their right-hand side is 0 when the neuron is refractory (by multiplication with the `not_refractory` variable).

> > **Parameters eqs** : *Equations*

> > > The equations without refractory mechanism.

> > **Returns new_eqs** : *Equations*

> > > New equations, with added parameters and changed differential equations having the "unless refractory" flag.

check_identifier_refractory(identifier)	Check that the identifier is not using a name reserved for the refractory mechanism.

check_identifier_refractory function

(*Shortest import*: `from brian2.equations.refractory import check_identifier_refractory`)

`brian2.equations.refractory.`**`check_identifier_refractory`**(*identifier*)

Check that the identifier is not using a name reserved for the refractory mechanism. The reserved names are `not_refractory, refractory, refractory_until`.

> **Parameters identifier** : str
>
> > The identifier to check.

> ### Raises

> **`ValueError`** If the identifier is a variable name used for the refractory mechanism.

`unitcheck` **module**

Utility functions for handling the units in *Equations*.

Exported members: *check_dimensions*, *check_units_statements*

Functions

check_dimensions(expression, dimensions, ...)	Compares the physical dimensions of an expression to expected dimensions in a given namespace.

check_dimensions function

(*Shortest import*: `from brian2.equations.unitcheck import check_dimensions`)

`brian2.equations.unitcheck.`**`check_dimensions`**(*expression*, *dimensions*, *variables*)

Compares the physical dimensions of an expression to expected dimensions in a given namespace.

> **Parameters expression** : str
>
> > The expression to evaluate.
>
> **dimensions** : `Dimension`
>
> > The expected physical dimensions for the `expression`.
>
> **variables** : dict
>
> > Dictionary of all variables (including external constants) used in the `expression`.

Raises

KeyError In case on of the identifiers cannot be resolved.

DimensionMismatchError If an unit mismatch occurs during the evaluation.

check_units_statements(code, variables)	Check the units for a series of statements.

check_units_statements function

(*Shortest import*: `from brian2.equations.unitcheck import check_units_statements`)

`brian2.equations.unitcheck.`**`check_units_statements`**(*code*, *variables*)

Check the units for a series of statements. Setting a model variable has to use the correct unit. For newly introduced temporary variables, the unit is determined and used to check the following statements to ensure consistency.

Parameters **code** : str

The statements as a (multi-line) string

variables : dict of `Variable` objects

The information about all variables used in `code` (including `Constant` objects for external variables)

Raises

KeyError In case on of the identifiers cannot be resolved.

DimensionMismatchError If an unit mismatch occurs during the evaluation.

6.7.5 groups package

Package providing groups such as *NeuronGroup* or *PoissonGroup*.

Exported members: *CodeRunner*, *Group*, *VariableOwner*, *NeuronGroup*

group module

This module defines the *VariableOwner* class, a mix-in class for everything that saves state variables, e.g. *Clock* or *NeuronGroup*, the class *Group* for objects that in addition to storing state variables also execute code, i.e. objects such as *NeuronGroup* or *StateMonitor* but not *Clock*, and finally *CodeRunner*, a class to run code in the context of a *Group*.

Exported members: *Group*, *VariableOwner*, *CodeRunner*

Classes

CodeRunner(*args, **kw)	A "code runner" that runs a CodeObject every timestep and keeps a reference to the *Group*.

CodeRunner class

(*Shortest import*: `from brian2 import CodeRunner`)

class `brian2.groups.group.`**`CodeRunner`**(**args*, ***kw*)

    Bases: *`brian2.core.base.BrianObject`*

A "code runner" that runs a `CodeObject` every timestep and keeps a reference to the *`Group`*. Used in *`NeuronGroup`* for `Thresholder`, `Resetter` and `StateUpdater`.

On creation, we try to run the before_run method with an empty additional namespace (see *`Network.before_run`*). If the namespace is already complete this might catch unit mismatches.

    Parameters **group** : *`Group`*

        The group to which this object belongs.

    template : `Template`

        The template that should be used for code generation

    code : str, optional

        The abstract code that should be executed every time step. The *`update_abstract_code`* method might generate this code dynamically before every run instead.

    dt : *`Quantity`*, optional

        The time step to be used for the simulation. Cannot be combined with the `clock` argument.

    user_code : str, optional

        The abstract code as specified by the user, i.e. without any additions of internal code that the user not necessarily knows about. This will be used for warnings and error messages.

    clock : *`Clock`*, optional

        The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *`defaultclock`* will be used.

    when : str, optional

        In which scheduling slot to execute the operation during a time step. Defaults to `'start'`.

    order : int, optional

        The priority of this operation for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

    name : str, optional

        The name for this object.

    check_units : bool, optional

        Whether the units should be checked for consistency before a run. Is activated (`True`) by default but should be switched off for state updaters (units are already checked for the equations and the generated abstract code might have already replaced variables with their unit-less values)

    template_kwds : dict, optional

        A dictionary of additional information that is passed to the template.

needed_variables: list of str, optional :

> A list of variables that are neither present in the abstract code, nor in the USES_VARIABLES statement in the template. This is only rarely necessary, an example being a *StateMonitor* where the names of the variables are neither known to the template nor included in the abstract code statements.

override_conditional_write: list of str, optional :

> A list of variable names which are used as conditions (e.g. for refractoriness) which should be ignored.

codeobj_class : class, optional

> The CodeObject class to run code with. If not specified, defaults to the group's codeobj_class attribute.

generate_empty_code : bool, optional

> Whether to generate a CodeObject if there is no abstract code to execute. Defaults to True but should be switched off e.g. for a StateUpdater when there is nothing to do.

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.
create_code_objects(run_namespace)	
create_default_code_object(run_namespace)	
update_abstract_code(run_namespace)	Update the abstract code for the code object.

Details

before_run(*run_namespace*)
> Optional method to prepare the object before a run.
>
> Called by *Network.after_run* before the main simulation loop starts.

create_code_objects(*run_namespace*)

create_default_code_object(*run_namespace*)

update_abstract_code(*run_namespace*)
> Update the abstract code for the code object. Will be called in *before_run* and should update the CodeRunner.abstract_code attribute.
>
> Does nothing by default.

Group(*args, **kw)

Methods

Group class

(*Shortest import*: `from brian2 import Group`)

class brian2.groups.group.**Group**(**args*, ***kw*)
  Bases: *[brian2.groups.group.VariableOwner](), [brian2.core.base.BrianObject]()*

Methods

[custom_operation]()(*args, **kwds)	

[resolve_all]()(identifiers, run_namespace[, ...])	Resolve a list of identifiers.
[run_regularly]()(code[, dt, clock, when, ...])	Run abstract code in the group's namespace.
[runner]()(*args, **kwds)	

Details

custom_operation(**args*, ***kwds*)

resolve_all(*identifiers*, *run_namespace*, *user_identifiers=None*, *additional_variables=None*)
  Resolve a list of identifiers. Calls `Group._resolve` for each identifier.

  Parameters   **identifiers** : iterable of str

   The names to look up.

   run_namespace : dict-like, optional

   An additional namespace that is used for variable lookup (if not defined, the implicit namespace of local variables is used).

   user_identifiers : iterable of str, optional

   The names in `identifiers` that were provided by the user (i.e. are part of user-specified equations, abstract code, etc.). Will be used to determine when to issue namespace conflict warnings. If not specified, will be assumed to be identical to `identifiers`.

   additional_variables : dict-like, optional

   An additional mapping of names to `Variable` objects that will be checked before `Group.variables`.

  Returns   **variables** : dict of `Variable` or *[Function]()*

   A mapping from name to `Variable`/*[Function]()* object for each of the names given in `identifiers`

Raises

> **KeyError** If one of the names in `identifier` cannot be resolved

run_regularly(*code*, *dt=None*, *clock=None*, *when='start'*, *order=0*, *name=None*, *codeobj_class=None*)
> Run abstract code in the group's namespace. The created *CodeRunner* object will be automatically added to the group, it therefore does not need to be added to the network manually. However, a reference to the object will be returned, which can be used to later remove it from the group or to set it to inactive.

> > **Parameters** **code** : str
> >
> > > The abstract code to run.
> >
> > **dt** : *Quantity*, optional
> >
> > > The time step to use for this custom operation. Cannot be combined with the `clock` argument.
> >
> > **clock** : *Clock*, optional
> >
> > > The update clock to use for this operation. If neither a clock nor the `dt` argument is specified, defaults to the clock of the group.
> >
> > **when** : str, optional
> >
> > > When to run within a time step, defaults to the `'start'` slot.
> >
> > **name** : str, optional
> >
> > > A unique name, if non is given the name of the group appended with 'run_regularly', 'run_regularly_1', etc. will be used. If a name is given explicitly, it will be used as given (i.e. the group name will not be prepended automatically).
> >
> > **codeobj_class** : class, optional
> >
> > > The `CodeObject` class to run code with. If not specified, defaults to the `group`'s `codeobj_class` attribute.
> >
> > **Returns** **obj** : *CodeRunner*
> >
> > > A reference to the object that will be run.

> **runner**(**args*, ***kwds*)

IndexWrapper(group)	Convenience class to allow access to the indices via indexing syntax.

IndexWrapper class

(*Shortest import*: `from brian2.groups.group import IndexWrapper`)

class `brian2.groups.group.`**IndexWrapper**(*group*)
> Bases: `object`

> Convenience class to allow access to the indices via indexing syntax. This allows for example to get all indices for synapses originating from neuron 10 by writing `synapses.indices[10, :]` instead of `synapses._indices.((10, slice(None))`.

[*Indexing*](group[, default_index])	Object responsible for calculating flat index arrays from arbitrary group- specific indices.

Indexing class

(*Shortest import*: `from brian2.groups.group import Indexing`)

class brian2.groups.group.**Indexing**(*group*, *default_index='_idx'*)
    Bases: [`object`](#)

    Object responsible for calculating flat index arrays from arbitrary group- specific indices. Stores strong references to the necessary variables so that basic indexing (i.e. slicing, integer arrays/values, …) works even when the respective [*VariableOwner*](#) no longer exists. Note that this object does not handle string indexing.

Methods

[*__call__*](#)([item, index_var])	Return flat indices to index into state variables from arbitrary group specific indices.

Details

__call__(*item=slice(None, None, None)*, *index_var=None*)
    Return flat indices to index into state variables from arbitrary group specific indices. In the default implementation, raises an error for multidimensional indices and transforms slices into arrays.

        Parameters item : slice, array, int

            The indices to translate.

        Returns indices : [`numpy.ndarray`](#)

            The flat indices corresponding to the indices given in `item`.

    See also:

    SynapticIndexing

[*VariableOwner*](#)(*args, **kw)	Mix-in class for accessing arrays by attribute.

VariableOwner class

(*Shortest import*: `from brian2 import VariableOwner`)

class brian2.groups.group.**VariableOwner**(**args*, ***kw*)
    Bases: [*brian2.core.names.Nameable*](#)

    Mix-in class for accessing arrays by attribute.

    # TODO: Overwrite the __dir__ method to return the state variables # (should make autocompletion work)

Methods

add_attribute(name)	Add a new attribute to this group.
check_variable_write(variable)	Function that can be overwritten to raise an error if writing to a variable should not be allowed.
get_states([vars, units, format, ...])	Return a copy of the current state variable values.
set_states(values[, units, format, level])	Set the state variables.
state(name[, use_units, level])	Return the state variable in a way that properly supports indexing in the context of this group

Details

add_attribute(*name*)

Add a new attribute to this group. Using this method instead of simply assigning to the new attribute name is necessary because Brian will raise an error in that case, to avoid bugs passing unnoticed (misspelled state variable name, un-declared state variable, …).

> **Parameters name** : str
>
> > The name of the new attribute

Raises

> **AttributeError** If the name already exists as an attribute or a state variable.

check_variable_write(*variable*)

Function that can be overwritten to raise an error if writing to a variable should not be allowed. Note that this does *not* deal with incorrect writes that are general to all kind of variables (incorrect units, writing to a read-only variable, etc.). This function is only used for type-specific rules, e.g. for raising an error in *Synapses* when writing to a synaptic variable before any `connect` call.

By default this function does nothing.

> **Parameters variable** : `Variable`
>
> > The variable that the user attempts to set.

get_states(*vars=None*, *units=True*, *format='dict'*, *subexpressions=False*, *read_only_variables=True*, *level=0*)

Return a copy of the current state variable values. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

> **Parameters vars** : list of str, optional
>
> > The names of the variables to extract. If not specified, extract all state variables (except for internal variables, i.e. names that start with `'_'`). If the `subexpressions` argument is `True`, the current values of all subexpressions are returned as well.
>
> **units** : bool, optional
>
> > Whether to include the physical units in the return value. Defaults to `True`.
>
> **format** : str, optional
>
> > The output format. Defaults to `'dict'`.

subexpressions: bool, optional :

    Whether to return subexpressions when no list of variable names is given. Defaults to `False`. This argument is ignored if an explicit list of variable names is given in `vars`.

read_only_variables : bool, optional

    Whether to return read-only variables (e.g. the number of neurons, the time, etc.). Setting it to `False` will assure that the returned state can later be used with *set_states*. Defaults to `True`.

level : int, optional

    How much higher to go up the stack to resolve external variables. Only relevant if extracting subexpressions that refer to external variables.

Returns values : dict or specified format

    The variables specified in `vars`, in the specified `format`.

set_states(*values*, *units=True*, *format='dict'*, *level=0*)

    Set the state variables.

    Parameters values : depends on `format`

        The values according to `format`.

    units : bool, optional

        Whether the `values` include physical units. Defaults to `True`.

    format : str, optional

        The format of `values`. Defaults to `'dict'`

    level : int, optional

        How much higher to go up the stack to resolve external variables. Only relevant when using string expressions to set values.

state(*name*, *use_units=True*, *level=0*)

    Return the state variable in a way that properly supports indexing in the context of this group

    Parameters name : str

        The name of the state variable

    use_units : bool, optional

        Whether to use the state variable's unit.

    level : int, optional

        How much farther to go down in the stack to find the namespace.

    Returns :

    ——- :

    var : `VariableView` or scalar value

        The state variable's value that can be indexed (for non-scalar values).

Functions

get_dtype(equation[, dtype])	Helper function to interpret the dtype keyword argument in *NeuronGroup* etc.

get_dtype function

(*Shortest import*: `from brian2.groups.group import get_dtype`)

brian2.groups.group.**get_dtype**(*equation*, *dtype=None*)

    Helper function to interpret the dtype keyword argument in *NeuronGroup* etc.

        Parameters equation : `SingleEquation`

            The equation for which a dtype should be returned

        dtype : dtype or dict, optional

            Either the dtype to be used as a default dtype for all float variables (instead of the *core.default_float_dtype* preference) or a dictionary stating the dtype for some variables; all other variables will use the preference default

        Returns d : dtype

            The dtype for the variable defined in `equation`

neurongroup module

This model defines the *NeuronGroup*, the core of most simulations.

Exported members: NeuronGroup

Classes

NeuronGroup(*args, **kw)	A group of neurons.

NeuronGroup class

(*Shortest import*: `from brian2 import NeuronGroup`)

class brian2.groups.neurongroup.**NeuronGroup**(**args*, ***kw*)

    Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

    A group of neurons.

        Parameters N : int

            Number of neurons in the group.

        model : str, *Equations*

            The differential equations defining the group

        method : (str, function), optional

            The numerical integration method. Either a string with the name of a registered method (e.g. "euler") or a function that receives an *Equations* object and returns the corresponding abstract code. If no method is specified, a suitable method will be chosen automatically.

threshold : str, optional

> The condition which produces spikes. Should be a single line boolean expression.

reset : str, optional

> The (possibly multi-line) string with the code to execute on reset.

refractory : {str, *Quantity*}, optional

> Either the length of the refractory period (e.g. `2*ms`), a string expression that evaluates to the length of the refractory period after each spike (e.g. `'(1 + rand())*ms'`), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. `'v > -20*mV'`)

events : dict, optional

> User-defined events in addition to the "spike" event defined by the `threshold`. Has to be a mapping of strings (the event name) to strings (the condition) that will be checked.

namespace: dict, optional :

> A dictionary mapping identifier names to objects. If not given, the namespace will be filled in at the time of the call of *Network.run*, with either the values from the `namespace` argument of the *Network.run* method or from the local context, if no such argument is given.

dtype : (`dtype`, `dict`), optional

> The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

codeobj_class : class, optional

> The `CodeObject` class to run code with.

dt : *Quantity*, optional

> The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

> The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

order : int, optional

> The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

> A unique name for the group, otherwise use `neurongroup_0`, etc.

Notes

NeuronGroup contains a *StateUpdater*, *Thresholder* and *Resetter*, and these are run at the 'groups', 'thresholds' and 'resets' slots (i.e. the values of their `when` attribute take these values). The `order` attribute will be passed down to the contained objects but can be set individually by setting the `order` attribute of the *state_updater*, *thresholder* and *resetter* attributes, respectively.

Attributes

_refractory	The refractory condition or timespan
event_codes	Code that is triggered on events (e.g.
events	Events supported by this group
method_choice	The state update method selected by the user
resetter	Reset neurons which have spiked (or perform arbitrary actions for user-defined events)
spikes	The spikes returned by the most recent thresholding operation.
state_updater	Performs numerical integration step
subexpression_updater	Update the "constant over a time step" subexpressions
thresholder	Checks the spike threshold (or abitrary user-defined events)
user_equations	The original equations as specified by the user (i.e.

Methods

before_run([run_namespace])	Optional method to prepare the object before a run.
run_on_event(event, code[, when, order])	Run code triggered by a custom-defined event (see *NeuronGroup* documentation for the specification of events).The created *Resetter* object will be automatically added to the group, it therefore does not need to be added to the network manually.
set_event_schedule(event[, when, order])	Change the scheduling slot for checking the condition of an event.
state(name[, use_units, level])	Return the state variable in a way that properly supports indexing in the context of this group

Details

_refractory
> The refractory condition or timespan

event_codes
> Code that is triggered on events (e.g. reset)

events
> Events supported by this group

method_choice
> The state update method selected by the user

resetter
> Reset neurons which have spiked (or perform arbitrary actions for user-defined events)

spikes
> The spikes returned by the most recent thresholding operation.

state_updater
> Performs numerical integration step

subexpression_updater
> Update the "constant over a time step" subexpressions

thresholder
> Checks the spike threshold (or abitrary user-defined events)

user_equations
> The original equations as specified by the user (i.e. without the multiplied `int(not_refractory)` term for equations marked as `(unless refractory)`)

before_run (*run_namespace=None*)
> Optional method to prepare the object before a run.
>
> Called by *Network.after_run* before the main simulation loop starts.

run_on_event (*event*, *code*, *when='after_resets'*, *order=None*)
> Run code triggered by a custom-defined event (see *NeuronGroup* documentation for the specification of events).The created *Resetter* object will be automatically added to the group, it therefore does not need to be added to the network manually. However, a reference to the object will be returned, which can be used to later remove it from the group or to set it to inactive.
>
> > **Parameters event** : str
> >
> > > The name of the event that should trigger the code
> >
> > **code** : str
> >
> > > The code that should be executed
> >
> > **when** : str, optional
> >
> > > The scheduling slot that should be used to execute the code. Defaults to `'after_resets'`.
> >
> > **order** : int, optional
> >
> > > The order for operations in the same scheduling slot. Defaults to the order of the *NeuronGroup*.
> >
> > **Returns obj** : *Resetter*
> >
> > > A reference to the object that will be run.

set_event_schedule (*event*, *when='after_thresholds'*, *order=None*)
> Change the scheduling slot for checking the condition of an event.
>
> > **Parameters event** : str
> >
> > > The name of the event for which the scheduling should be changed
> >
> > **when** : str, optional
> >
> > > The scheduling slot that should be used to check the condition. Defaults to `'after_thresholds'`.
> >
> > **order** : int, optional

> > The order for operations in the same scheduling slot. Defaults to the order of the *Neu-ronGroup*.

state(*name*, *use_units=True*, *level=0*)

> Return the state variable in a way that properly supports indexing in the context of this group

> > **Parameters** **name** : str

> > > The name of the state variable

> > **use_units** : bool, optional

> > > Whether to use the state variable's unit.

> > **level** : int, optional

> > > How much farther to go down in the stack to find the namespace.

> > **Returns** :

> > **——-** :

> > **var** : `VariableView` or scalar value

> > > The state variable's value that can be indexed (for non-scalar values).

Tutorials and examples using this

- Tutorial *1-intro-to-brian-neurons*
- Tutorial *2-intro-to-brian-synapses*
- Tutorial *3-intro-to-brian-simulations*
- Example *COBAHH*
- Example *CUBA*
- Example *IF_curve_Hodgkin_Huxley*
- Example *IF_curve_LIF*
- Example *adaptive_threshold*
- Example *advanced/COBAHH_approximated*
- Example *advanced/compare_GSL_to_conventional*
- Example *advanced/custom_events*
- Example *advanced/exprel_function*
- Example *advanced/float_32_64_benchmark*
- Example *advanced/modelfitting_sbi*
- Example *advanced/opencv_movie*
- Example *advanced/stochastic_odes*
- Example *compartmental/bipolar_with_inputs*
- Example *compartmental/bipolar_with_inputs2*
- Example *compartmental/lfp*
- Example *frompapers/Brette_2004*

- Example *frompapers/Brette_Gerstner_2005*

- Example *frompapers/Brette_Guigon_2003*

- Example *frompapers/Brunel_Hakim_1999*

- Example *frompapers/Brunel_Wang_2001*

- Example *frompapers/Clopath_et_al_2010_homeostasis*

- Example *frompapers/Clopath_et_al_2010_no_homeostasis*

- Example *frompapers/Diesmann_et_al_1999*

- Example *frompapers/Hindmarsh_Rose_1984*

- Example *frompapers/Izhikevich_2007*

- Example *frompapers/Jansen_Rit_1995_single_column*

- Example *frompapers/Kremer_et_al_2011_barrel_cortex*

- Example *frompapers/Morris_Lecar_1981*

- Example *frompapers/Platkiewicz_Brette_2011*

- Example *frompapers/Rossant_et_al_2011bis*

- Example *frompapers/Rothman_Manis_2003*

- Example *frompapers/Stimberg_et_al_2018/example_1_COBA*

- Example *frompapers/Stimberg_et_al_2018/example_2_gchi_astrocyte*

- Example *frompapers/Stimberg_et_al_2018/example_3_io_synapse*

- Example *frompapers/Stimberg_et_al_2018/example_4_rsmean*

- Example *frompapers/Stimberg_et_al_2018/example_4_synrel*

- Example *frompapers/Stimberg_et_al_2018/example_5_astro_ring*

- Example *frompapers/Stimberg_et_al_2018/example_6_COBA_with_astro*

- Example *frompapers/Sturzl_et_al_2000*

- Example *frompapers/Touboul_Brette_2008*

- Example *frompapers/Vogels_et_al_2011*

- Example *frompapers/Wang_Buszaki_1996*

- Example *multiprocessing/01_using_cython*

- Example *multiprocessing/02_using_standalone*

- Example *multiprocessing/03_standalone_joblib*

- Example *non_reliability*

- Example *phase_locking*

- Example *reliability*

- Example *standalone/STDP_standalone*

- Example *standalone/cuba_openmp*

- Example *standalone/simple_case*

- Example *standalone/simple_case_build*

- Example *standalone/standalone_multiplerun*

- Example *synapses/STDP*

- Example *synapses/continuous_interaction*

- Example *synapses/efficient_gaussian_connectivity*

- Example *synapses/gapjunctions*

- Example *synapses/jeffress*

- Example *synapses/licklider*

- Example *synapses/nonlinear*

- Example *synapses/spatial_connections*

- Example *synapses/spike_based_homeostasis*

- Example *synapses/state_variables*

- Example *synapses/synapses*

Resetter(*args, **kw)	The *CodeRunner* that applies the reset statement(s) to the state variables of neurons that have spiked in this timestep.

Resetter class

(*Shortest import*: `from brian2.groups.neurongroup import Resetter`)

class brian2.groups.neurongroup.**Resetter**(*args*, ***kw*)
    Bases: *brian2.groups.group.CodeRunner*

    The *CodeRunner* that applies the reset statement(s) to the state variables of neurons that have spiked in this timestep.

Methods

update_abstract_code(run_namespace)	Update the abstract code for the code object.

Details

update_abstract_code(*run_namespace*)
    Update the abstract code for the code object. Will be called in `before_run` and should update the `CodeRunner.abstract_code` attribute.

    Does nothing by default.

StateUpdater(*args, **kw)	The *CodeRunner* that updates the state variables of a *NeuronGroup* at every timestep.

StateUpdater class

(*Shortest import*: `from brian2.groups.neurongroup import StateUpdater`)

class brian2.groups.neurongroup.**StateUpdater**(**args*, ***kw*)
  Bases: *brian2.groups.group.CodeRunner*

  The *CodeRunner* that updates the state variables of a *NeuronGroup* at every timestep.

Methods

update_abstract_code(run_namespace)	Update the abstract code for the code object.

Details

update_abstract_code(*run_namespace*)
  Update the abstract code for the code object. Will be called in `before_run` and should update the
  `CodeRunner.abstract_code` attribute.

  Does nothing by default.

SubexpressionUpdater(*args, **kw)	The *CodeRunner* that updates the state variables storing the values of subexpressions that have been marked as "constant over dt".

SubexpressionUpdater class

(*Shortest import*: `from brian2.groups.neurongroup import SubexpressionUpdater`)

class brian2.groups.neurongroup.**SubexpressionUpdater**(**args*, ***kw*)
  Bases: *brian2.groups.group.CodeRunner*

  The *CodeRunner* that updates the state variables storing the values of subexpressions that have been marked as
  "constant over dt".

Thresholder(*args, **kw)	The *CodeRunner* that applies the threshold condition to the state variables of a *NeuronGroup* at every timestep and sets its `spikes` and `refractory_until` attributes.

Thresholder class

(*Shortest import*: `from brian2.groups.neurongroup import Thresholder`)

class brian2.groups.neurongroup.**Thresholder**(**args*, ***kw*)
  Bases: *brian2.groups.group.CodeRunner*

  The *CodeRunner* that applies the threshold condition to the state variables of a *NeuronGroup* at every timestep
  and sets its `spikes` and `refractory_until` attributes.

Methods

update_abstract_code(run_namespace)	Update the abstract code for the code object.

Details

update_abstract_code(*run_namespace*)

Update the abstract code for the code object. Will be called in `before_run` and should update the `CodeRunner.abstract_code` attribute.

Does nothing by default.

Functions

check_identifier_pre_post(identifier)	Do not allow names ending in _pre or _post to avoid confusion.

check_identifier_pre_post function

(*Shortest import*: `from brian2.groups.neurongroup import check_identifier_pre_post`)

brian2.groups.neurongroup.**check_identifier_pre_post**(*identifier*)

Do not allow names ending in _pre or _post to avoid confusion.

to_start_stop(item, N)	Helper function to transform a single number, a slice or an array of contiguous indices to a start and stop value.

to_start_stop function

(*Shortest import*: `from brian2.groups.neurongroup import to_start_stop`)

brian2.groups.neurongroup.**to_start_stop**(*item*, *N*)

Helper function to transform a single number, a slice or an array of contiguous indices to a start and stop value. This is used to allow for some flexibility in the syntax of specifying subgroups in *NeuronGroup* and *Spatial-Neuron*.

Parameters **item** : slice, int or sequence

The slice, index, or sequence of indices to use. Note that a sequence of indices has to be a sorted ascending sequence of subsequent integers.

N : int

The total number of elements in the group.

Returns **start** : int

The start value of the slice.

stop : int

The stop value of the slice.

Examples

```
>>> from brian2.groups.neurongroup import to_start_stop
>>> to_start_stop(slice(3, 6), 10)
(3, 6)
>>> to_start_stop(slice(3, None), 10)
(3, 10)
>>> to_start_stop(5, 10)
(5, 6)
>>> to_start_stop([3, 4, 5], 10)
(3, 6)
>>> to_start_stop([3, 5, 7], 10)
Traceback (most recent call last):
    ...
IndexError: Subgroups can only be constructed using contiguous indices.
```

`subgroup` module

Exported members: *Subgroup*

Classes

Subgroup(*args, **kw)	Subgroup of any *Group*

Subgroup class

(*Shortest import*: `from brian2 import Subgroup`)

class brian2.groups.subgroup.**Subgroup**(**args*, ***kw*)

    Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

    Subgroup of any *Group*

> **Parameters** **source** : SpikeSource
>
> > The source object to subgroup.
>
> **start, stop** : int
>
> > Select only spikes with indices from `start` to `stop-1`.
>
> **name** : str, optional
>
> > A unique name for the group, or use `source.name+'_subgroup_0'`, etc.

Attributes

spikes	

Details

spikes

6.7.6 importexport package

Package providing import/export support.

Exported members: `ImportExport`

`dictlike` module

Module providing `DictImportExport` and `PandasImportExport` (requiring a working installation of pandas).

Classes

`DictImportExport()`	An importer/exporter for variables in format of dict of numpy arrays.

DictImportExport class

(*Shortest import*: `from brian2.importexport import DictImportExport`)

class `brian2.importexport.dictlike.`**`DictImportExport`**
    Bases: `brian2.importexport.importexport.ImportExport`

    An importer/exporter for variables in format of dict of numpy arrays.

Attributes

`name`	Abstract property giving a method name.

Methods

`export_data`(group, variables[, units, level])	Asbtract static export data method with two obligatory parameters.
`import_data`(group, data[, units, level])	Import and set state variables.

Details

name

static export_data (*group*, *variables*, *units=True*, *level=0*)
    Asbtract static export data method with two obligatory parameters. It should return a copy of the current state variable values. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

        Parameters group : `Group`

Group object.

variables : list of str

The names of the variables to extract.

static import_data (*group*, *data*, *units=True*, *level=0*)
    Import and set state variables.

        Parameters group : [*Group*](#)

           Group object.

        data : dict_like

           Data to import with variable names.

[*PandasImportExport*](#)()	An importer/exporter for variables in pandas DataFrame format.

PandasImportExport class

(*Shortest import*: `from brian2.importexport import PandasImportExport`)

class brian2.importexport.dictlike.**PandasImportExport**
    Bases: [*brian2.importexport.importexport.ImportExport*](#)

    An importer/exporter for variables in pandas DataFrame format.

Attributes

[*name*](#)	Abstract property giving a method name.

Methods

[*export_data*](#)(group, variables[, units, level])	Asbtract static export data method with two obligatory parameters.
[*import_data*](#)(group, data[, units, level])	Import and set state variables.

Details

name

static export_data (*group*, *variables*, *units=True*, *level=0*)
    Asbtract static export data method with two obligatory parameters. It should return a copy of the current state variable values. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

        Parameters group : [*Group*](#)

           Group object.

        variables : list of str

           The names of the variables to extract.

static import_data (*group*, *data*, *units=True*, *level=0*)

    Import and set state variables.

        Parameters group : *Group*

            Group object.

        data : dict_like

            Data to import with variable names.

importexport module

Module defining the *ImportExport* class that enables getting state variable data in and out of groups in various formats (see `Group.get_states` and `Group.set_states`).

Classes

ImportExport()	Class for registering new import/export methods (via static methods).

ImportExport class

(*Shortest import*: `from brian2 import ImportExport`)

class `brian2.importexport.importexport.`**ImportExport**

    Bases: `object`

    Class for registering new import/export methods (via static methods). Also the base class that should be extended for such methods (*ImportExport.export_data*, *ImportExport.import_data*, and *ImportExport.name* have to be overwritten).

    See also:

    VariableOwner.get_states, *VariableOwner.set_states*

Attributes

methods	A dictionary mapping import/export methods names to *ImportExport* objects
name	Abstract property giving a method name.

Methods

export_data(group, variables)	Asbtract static export data method with two obligatory parameters.
import_data(group, data)	Import and set state variables.
register(importerexporter)	Register a import/export method.

Details

methods
>   A dictionary mapping import/export methods names to *ImportExport* objects

name
>   Abstract property giving a method name.

abstract static export_data(*group*, *variables*)
>   Asbtract static export data method with two obligatory parameters. It should return a copy of the current state variable values. The returned arrays are copies of the actual arrays that store the state variable values, therefore changing the values in the returned dictionary will not affect the state variables.

>>   **Parameters group** : *Group*

>>>   Group object.

>>   **variables** : list of str

>>>   The names of the variables to extract.

abstract static import_data(*group*, *data*)
>   Import and set state variables.

>>   **Parameters group** : *Group*

>>>   Group object.

>>   **data** : dict_like

>>>   Data to import with variable names.

static register(*importerexporter*)
>   Register a import/export method. Registered methods can be referred to via their name.

>>   **Parameters importerexporter** : *ImportExport*

>>>   The importerexporter object, e.g. an `DictImportExport`.

6.7.7 input package

Classes for providing external input to a network.

Exported members: *BinomialFunction*, *PoissonGroup*, *PoissonInput*, *SpikeGeneratorGroup*, *TimedArray*

binomial module

Implementation of *BinomialFunction*

Exported members: *BinomialFunction*

Classes

BinomialFunction(n, p[, approximate, name])	A function that generates samples from a binomial distribution.

BinomialFunction class

(*Shortest import*: `from brian2 import BinomialFunction`)

class brian2.input.binomial.**BinomialFunction**(*n*, *p*, *approximate=True*, *name='_binomial*'*)

Bases: *brian2.core.functions.Function*, *brian2.core.names.Nameable*

A function that generates samples from a binomial distribution.

> **Parameters** **n** : int
>
>> Number of samples
>
> **p** : float
>
>> Probablility
>
> **approximate** : bool, optional
>
>> Whether to approximate the binomial with a normal distribution if $np > 5 \wedge n(1-p) > 5$. Defaults to `True`.

Attributes

implementations	Container for implementing functions for different targets This container can be extended by other code-generation targets/devices The key has to be the name of the target, the value a function that takes three parameters (n, p, use_normal) and returns a tuple of (code, dependencies)

Details

implementations

Container for implementing functions for different targets This container can be extended by other code-generation targets/devices The key has to be the name of the target, the value a function that takes three parameters (n, p, use_normal) and returns a tuple of (code, dependencies)

`poissongroup` module

Implementation of *PoissonGroup*.

Exported members: *PoissonGroup*

Classes

PoissonGroup(*args, **kw)	Poisson spike source

PoissonGroup class

(*Shortest import*: `from brian2 import PoissonGroup`)

class brian2.input.poissongroup.**PoissonGroup**(**args*, ***kw*)
    Bases: *brian2.groups.group.Group*, *brian2.core.spikesource.SpikeSource*

    Poisson spike source

        Parameters **N** : int

            Number of neurons

        rates : *Quantity*, str

            Single rate, array of rates of length N, or a string expression evaluating to a rate. This
            string expression will be evaluated at every time step, it can therefore be time-dependent
            (e.g. refer to a *TimedArray*).

        dt : *Quantity*, optional

            The time step to be used for the simulation. Cannot be combined with the `clock` argu-
            ment.

        clock : *Clock*, optional

            The update clock to be used. If neither a clock, nor the `dt` argument is specified, the
            defaultclock will be used.

        when : str, optional

            When to run within a time step, defaults to the `'thresholds'` slot.

        order : int, optional

            The priority of of this group for operations occurring at the same time step and in the same
            scheduling slot. Defaults to 0.

        name : str, optional

            Unique name, or use poissongroup, poissongroup_1, etc.

Attributes

spikes	The spikes returned by the most recent thresholding operation.

Methods

before_run([run_namespace])	Optional method to prepare the object before a run.

Details

spikes
> The spikes returned by the most recent thresholding operation.

before_run(*run_namespace=None*)
> Optional method to prepare the object before a run.
>
> Called by *Network.after_run* before the main simulation loop starts.

Tutorials and examples using this

- Tutorial *3-intro-to-brian-simulations*
- Example *adaptive_threshold*
- Example *advanced/custom_events*
- Example *frompapers/Stimberg_et_al_2018/example_4_rsmean*
- Example *frompapers/Stimberg_et_al_2018/example_4_synrel*
- Example *standalone/STDP_standalone*
- Example *standalone/standalone_multiplerun*
- Example *synapses/STDP*
- Example *synapses/spike_based_homeostasis*

poissoninput module

Implementation of *PoissonInput*.

Exported members: *PoissonInput*

Classes

PoissonInput(target, target_var, N, rate, weight)	Adds independent Poisson input to a target variable of a *Group*.

PoissonInput class

(*Shortest import*: `from brian2 import PoissonInput`)

class brian2.input.poissoninput.**PoissonInput**(*target, target_var, N, rate, weight, when='synapses',*
order=0)
> Bases: *brian2.groups.group.CodeRunner*

> Adds independent Poisson input to a target variable of a *Group*. For large numbers of inputs, this is much more efficient than creating a *PoissonGroup*. The synaptic events are generated randomly during the simulation and are not preloaded and stored in memory. All the inputs must target the same variable, have the same frequency and same synaptic weight. All neurons in the target *Group* receive independent realizations of Poisson spike trains.

>> **Parameters target** : *Group*

>>> The group that is targeted by this input.

>> **target_var** : str

The variable of `target` that is targeted by this input.

N : int

The number of inputs

rate : *Quantity*

The rate of each of the inputs

weight : str or *Quantity*

Either a string expression (that can be interpreted in the context of `target`) or a *Quantity* that will be added for every event to the *target_var* of `target`. The unit has to match the unit of *target_var*

when : str, optional

When to update the target variable during a time step. Defaults to the `synapses` scheduling slot.

order : int, optional

The priority of of the update compared to other operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

Attributes

N	The number of inputs
rate	The rate of each input
target_var	The targetted variable
weight	The synaptic weight

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.

Details

N
The number of inputs

rate
The rate of each input

target_var
The targetted variable

weight
The synaptic weight

before_run(*run_namespace*)
Optional method to prepare the object before a run.

Called by *Network.after_run* before the main simulation loop starts.

Tutorials and examples using this

- Example *frompapers/Brunel_Wang_2001*

- Example *frompapers/Rossant_et_al_2011bis*

spikegeneratorgroup module

Module defining *SpikeGeneratorGroup*.

Exported members: *SpikeGeneratorGroup*

Classes

SpikeGeneratorGroup(N, indices, times[, dt, ...])	A group emitting spikes at given times.

SpikeGeneratorGroup class

(*Shortest import*: from brian2 import SpikeGeneratorGroup)

class brian2.input.spikegeneratorgroup.**SpikeGeneratorGroup**(*N, indices, times, dt=None, clock=None, period=0 * second, when='thresholds', order=0, sorted=False, name='spikegeneratorgroup*', codeobj_class=None*)

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*, *brian2.core. spikesource.SpikeSource*

A group emitting spikes at given times.

> **Parameters**   **N** : int
>
>> The number of "neurons" in this group
>
>> **indices** : array of integers
>
>> The indices of the spiking cells
>
>> **times** : *Quantity*
>
>> The spike times for the cells given in indices. Has to have the same length as indices.
>
>> **period** : *Quantity*, optional
>
>> If this is specified, it will repeat spikes with this period. A period of 0s means not repeating spikes.
>
>> **dt** : *Quantity*, optional
>
>> The time step to be used for the simulation. Cannot be combined with the clock argument.
>
>> **clock** : *Clock*, optional
>
>> The update clock to be used. If neither a clock, nor the dt argument is specified, the *defaultclock* will be used.
>
>> **when** : str, optional

When to run within a time step, defaults to the `'thresholds'` slot.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

sorted : bool, optional

Whether the given indices and times are already sorted. Set to `True` if your events are already sorted (first by spike time, then by index), this can save significant time at construction if your arrays contain large numbers of spikes. Defaults to `False`.

Notes

• If `sorted` is set to `True`, the given arrays will not be copied (only affects runtime mode)..

Attributes

_neuron_index	Array of spiking neuron indices.
_previous_dt	Remember the dt we used the last time when we checked the spike bins to not repeat the work for multiple runs with the same dt
_spike_time	Array of spiking neuron times.
_spikes_changed	"Dirty flag" that will be set when spikes are changed after the *before_run* check
spikes	The spikes returned by the most recent thresholding operation.

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.
set_spikes(indices, times[, period, sorted])	Change the spikes that this group will generate.

Details

_neuron_index
    Array of spiking neuron indices.

_previous_dt
    Remember the dt we used the last time when we checked the spike bins to not repeat the work for multiple runs with the same dt

_spike_time
    Array of spiking neuron times.

_spikes_changed
    "Dirty flag" that will be set when spikes are changed after the *before_run* check

spikes
    The spikes returned by the most recent thresholding operation.

before_run(*run_namespace*)

    Optional method to prepare the object before a run.

    Called by *Network.after_run* before the main simulation loop starts.

set_spikes(*indices*, *times*, *period=0 * second*, *sorted=False*)

    Change the spikes that this group will generate.

    This can be used to set the input for a second run of a model based on the output of a first run (if the input for the second run is already known before the first run, then all the information should simply be included in the initial *SpikeGeneratorGroup* initializer call, instead).

        Parameters **indices** : array of integers

            The indices of the spiking cells

            times : *Quantity*

            The spike times for the cells given in `indices`. Has to have the same length as `indices`.

            period : *Quantity*, optional

            If this is specified, it will repeat spikes with this period. A period of 0s means not repeating spikes.

            sorted : bool, optional

            Whether the given indices and times are already sorted. Set to `True` if your events are already sorted (first by spike time, then by index), this can save significant time at construction if your arrays contain large numbers of spikes. Defaults to `False`.

Tutorials and examples using this

- Tutorial *3-intro-to-brian-simulations*
- Example *frompapers/Diesmann_et_al_1999*
- Example *frompapers/Izhikevich_2007*
- Example *frompapers/Stimberg_et_al_2018/example_3_io_synapse*

timedarray module

Implementation of *TimedArray*.

Exported members: *TimedArray*

Classes

TimedArray(values, dt[, name])	A function of time built from an array of values.

TimedArray class

(*Shortest import*: `from brian2 import TimedArray`)

class `brian2.input.timedarray.`**TimedArray**(*values*, *dt*, *name=None*)

    Bases: *brian2.core.functions.Function*, *brian2.core.names.Nameable*, *brian2.utils.caching.CacheKey*

    A function of time built from an array of values. The returned object can be used as a function, including in model equations etc. The resulting function has to be called as `funcion_name(t)` if the provided value array is one-dimensional and as `function_name(t, i)` if it is two-dimensional.

        Parameters **values** : ndarray or *Quantity*

            An array of values providing the values at various points in time. This array can either be one- or two-dimensional. If it is two-dimensional it's first dimension should be the time.

        dt : *Quantity*

            The time distance between values in the `values` array.

        name : str, optional

            A unique name for this object, see *Nameable* for details. Defaults to `'_timedarray*'`.

Notes

For time values corresponding to elements outside of the range of `values` provided, the first respectively last element is returned.

Examples

```
>>> from brian2 import *
>>> ta = TimedArray([1, 2, 3, 4] * mV, dt=0.1*ms)
>>> print(ta(0.3*ms))
4. mV
>>> G = NeuronGroup(1, 'v = ta(t) : volt')
>>> mon = StateMonitor(G, 'v', record=True)
>>> net = Network(G, mon)
>>> net.run(1*ms)
...
>>> print(mon[0].v)
[ 1.  2.  3.  4.  4.  4.  4.  4.  4.  4.] mV
>>> ta2d = TimedArray([[1, 2], [3, 4], [5, 6]]*mV, dt=0.1*ms)
>>> G = NeuronGroup(4, 'v = ta2d(t, i%2) : volt')
>>> mon = StateMonitor(G, 'v', record=True)
>>> net = Network(G, mon)
>>> net.run(0.2*ms)
...
>>> print(mon.v[:])
[[ 1.  3.]
 [ 2.  4.]
 [ 1.  3.]
 [ 2.  4.]] mV
```

Attributes

implementations	Container for implementing functions for different targets This container can be extended by other code-generation targets/devices The key has to be the name of the target, the value is a tuple of functions, the first for a 1d array, the second for a 2d array.

Methods

is_locally_constant(dt)	Return whether this function (if interpreted as a function of time) should be considered constant over a timestep.

Details

implementations

Container for implementing functions for different targets This container can be extended by other codegeneration targets/devices The key has to be the name of the target, the value is a tuple of functions, the first for a 1d array, the second for a 2d array. The functions have to take three parameters: (values, dt, name), i.e. the array values, their physical dimensions, the dt of the TimedArray, and the name of the TimedArray. The functions have to return *a function* that takes the `owner` argument (out of which they can get the context's dt as `owner.clock.dt_`) and returns the code.

is_locally_constant(*dt*)

Return whether this function (if interpreted as a function of time) should be considered constant over a timestep. This is most importantly used by *TimedArray* so that linear integration can be used. In its standard implementation, always returns `False`.

> **Parameters dt** : float
>
> > The length of a timestep (without units).
>
> **Returns constant** : bool
>
> > Whether the results of this function can be considered constant over one timestep of length `dt`.

Tutorials and examples using this

- Tutorial *3-intro-to-brian-simulations*

- Example *advanced/COBAHH_approximated*

- Example *frompapers/Brunel_Wang_2001*

- Example *frompapers/Jansen_Rit_1995_single_column*

- Example *frompapers/Stimberg_et_al_2018/example_4_synrel*

- Example *frompapers/Stimberg_et_al_2018/example_6_COBA_with_astro*

- Example *frompapers/Sturzl_et_al_2000*

- Example *synapses/jeffress*

- Example *synapses/spike_based_homeostasis*

6.7.8 memory package

dynamicarray module

TODO: rewrite this (verbatim from Brian 1.x), more efficiency

Exported members: `DynamicArray`, `DynamicArray1D`

Classes

DynamicArray(shape[, dtype, factor, ...])	An N-dimensional dynamic array class

DynamicArray class

(*Shortest import*: `from brian2.memory.dynamicarray import DynamicArray`)

class brian2.memory.dynamicarray.**DynamicArray**(*shape*, *dtype=<class 'float'>*, *factor=2*, *use_numpy_resize=False*, *refcheck=True*)

Bases: `object`

An N-dimensional dynamic array class

The array can be resized in any dimension, and the class will handle allocating a new block of data and copying when necessary.

> **Warning:** The data will NOT be contiguous for >1D arrays. To ensure this, you will either need to use 1D arrays, or to copy the data, or use the shrink method with the current size (although note that in both cases you negate the memory and efficiency benefits of the dynamic array).

Initialisation arguments:

shape, dtype The shape and dtype of the array to initialise, as in Numpy. For 1D arrays, shape can be a single int, for ND arrays it should be a tuple.

factor The resizing factor (see notes below). Larger values tend to lead to more wasted memory, but more computationally efficient code.

use_numpy_resize, refcheck Normally, when you resize the array it creates a new array and copies the data. Sometimes, it is possible to resize an array without a copy, and if this option is set it will attempt to do this. However, this can cause memory problems if you are not careful so the option is off by default. You need to ensure that you do not create slices of the array so that no references to the memory exist other than the main array object. If you are sure you know what you're doing, you can switch this reference check off. Note that resizing in this way is only done if you resize in the first dimension.

The array is initialised with zeros. The data is stored in the attribute `data` which is a Numpy array.

Some numpy methods are implemented and can work directly on the array object, including `len(arr)`, `arr[..]` and `arr[...]=....` In other cases, use the `data` attribute.

Notes

The dynamic array returns a `data` attribute which is a view on the larger `_data` attribute. When a resize operation is performed, and a specific dimension is enlarged beyond the size in the `_data` attribute, the size is increased to the larger of `cursize*factor` and `newsize`. This ensures that the amortized cost of increasing the size of the array is O(1).

Examples

```
>>> x = DynamicArray((2, 3), dtype=int)
>>> x[:] = 1
>>> x.resize((3, 3))
>>> x[:] += 1
>>> x.resize((3, 4))
>>> x[:] += 1
>>> x.resize((4, 4))
>>> x[:] += 1
>>> x.data[:] = x.data**2
>>> x.data
array([[16, 16, 16,  4],
       [16, 16, 16,  4],
       [ 9,  9,  9,  4],
       [ 1,  1,  1,  1]])
```

Methods

resize(newshape)	Resizes the data to the new shape, which can be a different size to the current data, but should have the same rank, i.e. same number of dimensions.
resize_along_first(newshape)	
shrink(newshape)	Reduces the data to the given shape, which should be smaller than the current shape.

Details

resize(*newshape*)

Resizes the data to the new shape, which can be a different size to the current data, but should have the same rank, i.e. same number of dimensions.

resize_along_first(*newshape*)

shrink(*newshape*)

Reduces the data to the given shape, which should be smaller than the current shape. `resize()` can also be used with smaller values, but it will not shrink the allocated memory, whereas *shrink* will reallocate the memory. This method should only be used infrequently, as if it is used frequently it will negate the computational efficiency benefits of the DynamicArray.

DynamicArray1D(shape[, dtype, factor, ...])	Version of *DynamicArray* with specialised `resize` method designed to be more efficient.

DynamicArray1D class

(*Shortest import*: `from brian2.memory.dynamicarray import DynamicArray1D`)

class `brian2.memory.dynamicarray.`**`DynamicArray1D`** (*shape*, *dtype=<class 'float'>*, *factor=2*,
use_numpy_resize=False, *refcheck=True*)

    Bases: *[brian2.memory.dynamicarray.DynamicArray](#)*

    Version of *[DynamicArray](#)* with specialised `resize` method designed to be more efficient.

Methods

[resize](#)(newshape)	Resizes the data to the new shape, which can be a different size to the current data, but should have the same rank, i.e. same number of dimensions.

Details

resize (*newshape*)
    Resizes the data to the new shape, which can be a different size to the current data, but should have the same rank, i.e. same number of dimensions.

Functions

[getslices](#)(shape[, from_start])

getslices function

(*Shortest import*: `from brian2.memory.dynamicarray import getslices`)

`brian2.memory.dynamicarray.`**`getslices`** (*shape*, *from_start=True*)

6.7.9 monitors package

Base package for all monitors, i.e. objects to record activity during a simulation run.

Exported members: *[SpikeMonitor](#)*, *[EventMonitor](#)*, *[StateMonitor](#)*, *[PopulationRateMonitor](#)*

ratemonitor module

Module defining *[PopulationRateMonitor](#)*.

Exported members: *[PopulationRateMonitor](#)*

Classes

[PopulationRateMonitor](#)(*args, **kw)	Record instantaneous firing rates, averaged across neurons from a *[NeuronGroup](#)* or other spike source.

PopulationRateMonitor class

(*Shortest import*: `from brian2 import PopulationRateMonitor`)

class brian2.monitors.ratemonitor.**PopulationRateMonitor**(**args*, ***kw*)
    Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

    Record instantaneous firing rates, averaged across neurons from a *NeuronGroup* or other spike source.

> **Parameters source** : (*NeuronGroup*, *SpikeSource*)
>
>> The source of spikes to record.
>
> **name** : str, optional
>
>> A   unique   name   for   the   object,   otherwise   will   use   `source.`
>> `name+'_ratemonitor_0'`, etc.
>
> **codeobj_class** : class, optional
>
>> The `CodeObject` class to run code with.
>
> **dtype** : dtype, optional
>
>> The dtype to use to store the `rate` variable. Defaults to `float64`, i.e. double precision.
>
> **Notes** :
>
> **——** :
>
> **Currently, this monitor can only monitor the instantaneous firing rates at** :
>
> **each time step of the source clock. Any binning/smoothing of the firing** :
>
> **rates has to be done manually afterwards.** :

Attributes

source	The group we are recording from

Methods

reinit()	Clears all recorded rates
resize(new_size)	
smooth_rate(self[, window, width])	Return a smooth version of the population rate.

Details

source
    The group we are recording from

reinit()
    Clears all recorded rates

resize(*new_size*)

smooth_rate(*self*, *window='gaussian'*, *width=None*)
    Return a smooth version of the population rate.

        Parameters window : str, ndarray

            The window to use for smoothing. Can be a string to chose a predefined window(`'flat'` for a rectangular, and `'gaussian'` for a Gaussian-shaped window). In this case the width of the window is determined by the `width` argument. Note that for the Gaussian window, the `width` parameter specifies the standard deviation of the Gaussian, the width of the actual window is `4*width + dt` (rounded to the nearest dt). For the flat window, the width is rounded to the nearest odd multiple of dt to avoid shifting the rate in time. Alternatively, an arbitrary window can be given as a numpy array (with an odd number of elements). In this case, the width in units of time depends on the `dt` of the simulation, and no `width` argument can be specified. The given window will be automatically normalized to a sum of 1.

        width : *Quantity*, optional

            The width of the `window` in seconds (for a predefined window).

        Returns rate : *Quantity*

            The population rate in Hz, smoothed with the given window. Note that the rates are smoothed and not re-binned, i.e. the length of the returned array is the same as the length of the `rate` attribute and can be plotted against the *PopulationRateMonitor*'s t attribute.

Tutorials and examples using this

- Example *frompapers/Brunel_Hakim_1999*
- Example *frompapers/Brunel_Wang_2001*
- Example *synapses/spike_based_homeostasis*

`spikemonitor` **module**

Module defining *EventMonitor* and *SpikeMonitor*.

Exported members: *EventMonitor*, *SpikeMonitor*

Classes

EventMonitor(*args, **kw)	Record events from a *NeuronGroup* or another event source.

EventMonitor class

(*Shortest import*: `from brian2 import EventMonitor`)

class brian2.monitors.spikemonitor.**EventMonitor**(**args*, ***kw*)

Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

Record events from a *NeuronGroup* or another event source.

The recorded events can be accessed in various ways: the attributes `i` and `t` store all the indices and event times, respectively. Alternatively, you can get a dictionary mapping neuron indices to event trains, by calling the *event_trains* method.

> **Parameters** **source** : *NeuronGroup*, *SpikeSource*
>
> > The source of events to record.
>
> **event** : str
>
> > The name of the event to record
>
> **variables** : str or sequence of str, optional
>
> > Which variables to record at the time of the event (in addition to the index of the neuron). Can be the name of a variable or a list of names.
>
> **record** : bool, optional
>
> > Whether or not to record each event in `i` and `t` (the *count* will always be recorded). Defaults to `True`.
>
> **when** : str, optional
>
> > When to record the events, by default records events in the same slot where the event is emitted.
>
> **order** : int, optional
>
> > The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to the order where the event is emitted + 1, i.e. it will be recorded directly afterwards.
>
> **name** : str, optional
>
> > A unique name for the object, otherwise will use `source.name+'_eventmonitor_0'`, etc.
>
> **codeobj_class** : class, optional
>
> > The `CodeObject` class to run code with.

See also:

SpikeMonitor

Attributes

count	The array of event counts (length = size of target group)
event	The event that we are listening to
it	Returns the pair (`i`, `t`).
*it_*	Returns the pair (`i`, `t_`).
num_events	Returns the total number of recorded events.
record	Whether to record times and indices of events
record_variables	The additional variables that will be recorded
source	The source we are recording from

Methods

all_values()	Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time).
event_trains()	Return a dictionary mapping event indices to arrays of event times.
reinit()	Clears all recorded spikes
resize(new_size)	
values(var)	Return a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time).

Details

count
>   The array of event counts (length = size of target group)

event
>   The event that we are listening to

it
>   Returns the pair (`i`, `t`).

it_
>   Returns the pair (`i`, `t_`).

num_events
>   Returns the total number of recorded events.

record
>   Whether to record times and indices of events

record_variables
>   The additional variables that will be recorded

source
>   The source we are recording from

all_values()

Return a dictionary mapping recorded variable names (including `t`) to a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time). This is equivalent to (but more efficient than) calling `values` for each variable and storing the result in a dictionary.

> **Returns all_values** : dict
>
> > Dictionary mapping variable names to dictionaries which themselves are mapping neuron indicies to arrays of variable values at the time of the events.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, '''counter1 : integer
...                       counter2 : integer
...                       max_value : integer''',
...                 threshold='counter1 >= max_value',
...                 reset='counter1 = 0')
>>> G.run_regularly('counter1 += 1; counter2 += 1')
CodeRunner(...)
>>> G.max_value = [50, 100]
>>> mon = EventMonitor(G, event='spike', variables='counter2')
>>> run(10*ms)
>>> all_values = mon.all_values()
>>> print(all_values['counter2'][0])
[ 50 100]
>>> print(all_values['t'][1])
[ 9.9] ms
```

event_trains()

Return a dictionary mapping event indices to arrays of event times. Equivalent to calling `values('t')`.

> **Returns event_trains** : dict
>
> > Dictionary that stores an array with the event times for each neuron index.

> See also:
>
> *SpikeMonitor.spike_trains*

reinit()

Clears all recorded spikes

resize(*new_size*)

values(*var*)

Return a dictionary mapping neuron indices to arrays of variable values at the time of the events (sorted by time).

> **Parameters var** : str
>
> > The name of the variable.
>
> **Returns values** : dict
>
> > Dictionary mapping each neuron index to an array of variable values at the time of the events

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, '''counter1 : integer
...                       counter2 : integer
...                       max_value : integer''',
...                 threshold='counter1 >= max_value',
...                 reset='counter1 = 0')
>>> G.run_regularly('counter1 += 1; counter2 += 1')
CodeRunner(...)
>>> G.max_value = [50, 100]
>>> mon = EventMonitor(G, event='spike', variables='counter2')
>>> run(10*ms)
>>> counter2_values = mon.values('counter2')
>>> print(counter2_values[0])
[ 50 100]
>>> print(counter2_values[1])
[100]
```

Tutorials and examples using this

- Example *advanced/custom_events*

SpikeMonitor(*args, **kw)	Record spikes from a *NeuronGroup* or other spike source.

SpikeMonitor class

(*Shortest import*: `from brian2 import SpikeMonitor`)

class brian2.monitors.spikemonitor.**SpikeMonitor**(**args*, ***kw*)
    Bases: *brian2.monitors.spikemonitor.EventMonitor*

Record spikes from a *NeuronGroup* or other spike source.

The recorded spikes can be accessed in various ways (see Examples below): the attributes `i` and `t` store all the indices and spike times, respectively. Alternatively, you can get a dictionary mapping neuron indices to spike trains, by calling the *spike_trains* method. If you record additional variables with the `variables` argument, these variables can be accessed by their name (see Examples).

    Parameters **source** : (*NeuronGroup*, *SpikeSource*)

        The source of spikes to record.

    variables : str or sequence of str, optional

        Which variables to record at the time of the spike (in addition to the index of the neuron).
        Can be the name of a variable or a list of names.

    record : bool, optional

        Whether or not to record each spike in `i` and `t` (the *count* will always be recorded).
        Defaults to `True`.

    when : str, optional

When to record the events, by default records events in the same slot where the event is emitted.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to the order where the event is emitted + 1, i.e. it will be recorded directly afterwards.

name : str, optional

A unique name for the object, otherwise will use `source.name+'_spikemonitor_0'`, etc.

codeobj_class : class, optional

The `CodeObject` class to run code with.

Examples

```
>>> from brian2 import *
>>> spikes = SpikeGeneratorGroup(3, [0, 1, 2], [0, 1, 2]*ms)
>>> spike_mon = SpikeMonitor(spikes)
>>> net = Network(spikes, spike_mon)
>>> net.run(3*ms)
>>> print(spike_mon.i[:])
[0 1 2]
>>> print(spike_mon.t[:])
[ 0.  1.  2.] ms
>>> print(spike_mon.t_[:])
[ 0.     0.001  0.002]
>>> from brian2 import *
>>> G = NeuronGroup(2, '''counter1 : integer
...                       counter2 : integer
...                       max_value : integer''',
...                 threshold='counter1 >= max_value',
...                 reset='counter1 = 0')
>>> G.run_regularly('counter1 += 1; counter2 += 1')
CodeRunner(...)
>>> G.max_value = [50, 100]
>>> mon = SpikeMonitor(G, variables='counter2')
>>> net = Network(G, mon)
>>> net.run(10*ms)
>>> print(mon.i[:])
[0 0 1]
>>> print(mon.counter2[:])
[ 50 100 100]
```

Attributes

count	The array of spike counts (length = size of target group)
num_spikes	Returns the total number of recorded spikes.

Methods

all_values()	Return a dictionary mapping recorded variable names (including t) to a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time).
spike_trains()	Return a dictionary mapping spike indices to arrays of spike times.
values(var)	Return a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time).

Details

count
> The array of spike counts (length = size of target group)

num_spikes
> Returns the total number of recorded spikes.

all_values()
> Return a dictionary mapping recorded variable names (including t) to a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time). This is equivalent to (but more efficient than) calling *values* for each variable and storing the result in a dictionary.
>
> > **Returns all_values** : dict
> >
> > > Dictionary mapping variable names to dictionaries which themselves are mapping neuron indicies to arrays of variable values at the time of the spikes.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, '''counter1 : integer
...                       counter2 : integer
...                       max_value : integer''',
...                 threshold='counter1 >= max_value',
...                 reset='counter1 = 0')
>>> G.run_regularly('counter1 += 1; counter2 += 1')
CodeRunner(...)
>>> G.max_value = [50, 100]
>>> mon = SpikeMonitor(G, variables='counter2')
>>> run(10*ms)
>>> all_values = mon.all_values()
>>> print(all_values['counter2'][0])
```

(continues on next page)

```
[ 50 100]
>>> print(all_values['t'][1])
[ 9.9] ms
```

spike_trains()

> Return a dictionary mapping spike indices to arrays of spike times.

> > **Returns spike_trains** : dict

> > > Dictionary that stores an array with the spike times for each neuron index.

Examples

```
>>> from brian2 import *
>>> spikes = SpikeGeneratorGroup(3, [0, 1, 2], [0, 1, 2]*ms)
>>> spike_mon = SpikeMonitor(spikes)
>>> run(3*ms)
>>> spike_trains = spike_mon.spike_trains()
>>> spike_trains[1]
array([ 1.]) * msecond
```

values(*var*)

> Return a dictionary mapping neuron indices to arrays of variable values at the time of the spikes (sorted by time).

> > **Parameters var** : str

> > > The name of the variable.

> > **Returns values** : dict

> > > Dictionary mapping each neuron index to an array of variable values at the time of the spikes.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(2, '''counter1 : integer
...                       counter2 : integer
...                       max_value : integer''',
...                  threshold='counter1 >= max_value',
...                  reset='counter1 = 0')
>>> G.run_regularly('counter1 += 1; counter2 += 1')
CodeRunner(...)
>>> G.max_value = [50, 100]
>>> mon = SpikeMonitor(G, variables='counter2')
>>> run(10*ms)
>>> counter2_values = mon.values('counter2')
>>> print(counter2_values[0])
[ 50 100]
>>> print(counter2_values[1])
[100]
```

Tutorials and examples using this

- Tutorial *1-intro-to-brian-neurons*
- Tutorial *3-intro-to-brian-simulations*
- Example *CUBA*
- Example *IF_curve_Hodgkin_Huxley*
- Example *IF_curve_LIF*
- Example *adaptive_threshold*
- Example *advanced/custom_events*
- Example *advanced/modelfitting_sbi*
- Example *advanced/opencv_movie*
- Example *compartmental/hh_with_spikes*
- Example *frompapers/Brette_2004*
- Example *frompapers/Brette_2012/Fig5A*
- Example *frompapers/Brette_Gerstner_2005*
- Example *frompapers/Brette_Guigon_2003*
- Example *frompapers/Brunel_Hakim_1999*
- Example *frompapers/Brunel_Wang_2001*
- Example *frompapers/Diesmann_et_al_1999*
- Example *frompapers/Izhikevich_2007*
- Example *frompapers/Platkiewicz_Brette_2011*
- Example *frompapers/Rossant_et_al_2011bis*
- Example *frompapers/Stimberg_et_al_2018/example_1_COBA*
- Example *frompapers/Stimberg_et_al_2018/example_6_COBA_with_astro*
- Example *frompapers/Sturzl_et_al_2000*
- Example *frompapers/Touboul_Brette_2008*
- Example *frompapers/Vogels_et_al_2011*
- Example *non_reliability*
- Example *phase_locking*
- Example *reliability*
- Example *standalone/STDP_standalone*
- Example *standalone/cuba_openmp*
- Example *standalone/standalone_multiplerun*
- Example *synapses/STDP*
- Example *synapses/jeffress*
- Example *synapses/licklider*

statemonitor **module**

Exported members: *StateMonitor*

Classes

StateMonitor(*args, **kw)	Record values of state variables during a run

StateMonitor class

(*Shortest import*: from brian2 import StateMonitor)

class brian2.monitors.statemonitor.**StateMonitor**(**args*, ***kw*)

    Bases: *brian2.groups.group.Group*, *brian2.groups.group.CodeRunner*

    Record values of state variables during a run

    To extract recorded values after a run, use the t attribute for the array of times at which values were recorded, and variable name attribute for the values. The values will have shape (len(indices), len(t)), where indices are the array indices which were recorded. When indexing the *StateMonitor* directly, the returned object can be used to get the recorded values for the specified indices, i.e. the indexing semantic refers to the indices in source, not to the relative indices of the recorded values. For example, when recording only neurons with even numbers, mon[[0, 2]].v will return the values for neurons 0 and 2, whereas mon.v[[0, 2]] will return the values for the first and third *recorded* neurons, i.e. for neurons 0 and 4.

        Parameters   **source** : *Group*

            Which object to record values from.

        variables : str, sequence of str, True

            Which variables to record, or True to record all variables (note that this may use a great deal of memory).

        record : bool, sequence of ints

            Which indices to record, nothing is recorded for False, everything is recorded for True (warning: may use a great deal of memory), or a specified subset of indices.

        dt : *Quantity*, optional

            The time step to be used for the monitor. Cannot be combined with the clock argument.

        clock : *Clock*, optional

            The update clock to be used. If neither a clock, nor the dt argument is specified, the clock of the source() will be used.

        when : str, optional

            At which point during a time step the values should be recorded. Defaults to 'start'.

        order : int, optional

            The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

        name : str, optional

            A unique name for the object, otherwise will use source.name+'statemonitor_0', etc.

        codeobj_class : CodeObject, optional

The `CodeObject` class to create.

Notes

Since this monitor by default records in the `'start'` time slot, recordings of the membrane potential in integrate-and-fire models may look unexpected: the recorded membrane potential trace will never be above threshold in an integrate-and-fire model, because the reset statement will have been applied already. Set the `when` keyword to a different value if this is not what you want.

Note that `record=True` only works in runtime mode for synaptic variables. This is because the actual array of indices has to be calculated and this is not possible in standalone mode, where the synapses have not been created yet at this stage. Consider using an explicit array of indices instead, i.e. something like `record=np.arange(n_synapses)`.

Examples

Record all variables, first 5 indices:

```
eqs = '''
dV/dt = (2-V)/(10*ms) : 1
'''
threshold = 'V>1'
reset = 'V = 0'
G = NeuronGroup(100, eqs, threshold=threshold, reset=reset)
G.V = rand(len(G))
M = StateMonitor(G, True, record=range(5))
run(100*ms)
plot(M.t, M.V.T)
show()
```

Attributes

record	The array of recorded indices
record_variables	The variables to record

Methods

record_single_timestep()	Records a single time step.
reinit()	
resize(new_size)	

Details

record
> The array of recorded indices

record_variables
> The variables to record

record_single_timestep()
> Records a single time step. Useful for recording the values at the end of the simulation – otherwise a *State-Monitor* will not record the last simulated values since its `when` attribute defaults to `'start'`, i.e. the last recording is at the *beginning* of the last time step.

Notes

This function will only work if the *StateMonitor* has been already run, but a run with a length of `0*ms` does suffice.

Examples

```
>>> from brian2 import *
>>> G = NeuronGroup(1, 'dv/dt = -v/(5*ms) : 1')
>>> G.v = 1
>>> mon = StateMonitor(G, 'v', record=True)
>>> run(0.5*ms)
>>> print(np.array_str(mon.v[:], precision=3))
[[ 1.    0.98  0.961  0.942  0.923]]
>>> print(mon.t[:])
[   0.  100.  200.  300.  400.] us
>>> print(np.array_str(G.v[:], precision=3))  # last value had not been
→recorded
[ 0.905]
>>> mon.record_single_timestep()
>>> print(mon.t[:])
[   0.  100.  200.  300.  400.  500.] us
>>> print(np.array_str(mon.v[:], precision=3))
[[ 1.    0.98  0.961  0.942  0.923  0.905]]
```

reinit()

resize(*new_size*)

Tutorials and examples using this

- Tutorial *1-intro-to-brian-neurons*
- Tutorial *2-intro-to-brian-synapses*
- Tutorial *3-intro-to-brian-simulations*
- Example *COBAHH*
- Example *adaptive_threshold*
- Example *advanced/COBAHH_approximated*
- Example *advanced/compare_GSL_to_conventional*

- Example *advanced/custom_events*

- Example *advanced/modelfitting_sbi*

- Example *advanced/stochastic_odes*

- Example *compartmental/bipolar_cell*

- Example *compartmental/bipolar_with_inputs*

- Example *compartmental/bipolar_with_inputs2*

- Example *compartmental/hh_with_spikes*

- Example *compartmental/hodgkin_huxley_1952*

- Example *compartmental/infinite_cable*

- Example *compartmental/lfp*

- Example *compartmental/spike_initiation*

- Example *frompapers/Brette_2012/Fig1*

- Example *frompapers/Brette_2012/Fig3AB*

- Example *frompapers/Brette_2012/Fig3CF*

- Example *frompapers/Brette_2012/Fig4*

- Example *frompapers/Brette_2012/Fig5A*

- Example *frompapers/Brette_Gerstner_2005*

- Example *frompapers/Brette_Guigon_2003*

- Example *frompapers/Clopath_et_al_2010_homeostasis*

- Example *frompapers/Destexhe_et_al_1998*

- Example *frompapers/Hindmarsh_Rose_1984*

- Example *frompapers/Izhikevich_2007*

- Example *frompapers/Jansen_Rit_1995_single_column*

- Example *frompapers/Morris_Lecar_1981*

- Example *frompapers/Platkiewicz_Brette_2011*

- Example *frompapers/Rossant_et_al_2011bis*

- Example *frompapers/Rothman_Manis_2003*

- Example *frompapers/Stimberg_et_al_2018/example_1_COBA*

- Example *frompapers/Stimberg_et_al_2018/example_2_gchi_astrocyte*

- Example *frompapers/Stimberg_et_al_2018/example_3_io_synapse*

- Example *frompapers/Stimberg_et_al_2018/example_4_rsmean*

- Example *frompapers/Stimberg_et_al_2018/example_4_synrel*

- Example *frompapers/Stimberg_et_al_2018/example_5_astro_ring*

- Example *frompapers/Touboul_Brette_2008*

- Example *frompapers/Wang_Buszaki_1996*

- Example *multiprocessing/01_using_cython*

- Example *multiprocessing/02_using_standalone*

- Example *multiprocessing/03_standalone_joblib*

- Example *phase_locking*

- Example *standalone/STDP_standalone*

- Example *standalone/simple_case*

- Example *standalone/simple_case_build*

- Example *synapses/STDP*

- Example *synapses/continuous_interaction*

- Example *synapses/gapjunctions*

- Example *synapses/jeffress*

- Example *synapses/nonlinear*

- Example *synapses/spike_based_homeostasis*

- Example *synapses/synapses*

StateMonitorView(monitor, item)

StateMonitorView class

(*Shortest import*: `from brian2.monitors.statemonitor import StateMonitorView`)

class brian2.monitors.statemonitor.**StateMonitorView**(*monitor*, *item*)
    Bases: `object`

6.7.10 parsing package

`bast` module

Brian AST representation

This is a standard Python AST representation with additional information added.

Exported members: `brian_ast`, `BrianASTRenderer`, dtype_hierarchy

Classes

BrianASTRenderer(variables[, copy_variables])	This class is modelled after `NodeRenderer` - see there for details.

BrianASTRenderer class

(*Shortest import*: `from brian2.parsing.bast import BrianASTRenderer`)

class brian2.parsing.bast.**BrianASTRenderer**(*variables*, *copy_variables=True*)

    Bases: [object](#)

    This class is modelled after `NodeRenderer` - see there for details.

Methods

[*render_BinOp*](#)(node)
[*render_BoolOp*](#)(node)
[*render_Call*](#)(node)
[*render_Compare*](#)(node)
[*render_Constant*](#)(node)
[*render_Name*](#)(node)
[*render_NameConstant*](#)(node)
[*render_Num*](#)(node)
[*render_UnaryOp*](#)(node)
[*render_node*](#)(node)

Details

render_BinOp(*node*)

render_BoolOp(*node*)

render_Call(*node*)

render_Compare(*node*)

render_Constant(*node*)

render_Name(*node*)

render_NameConstant(*node*)

render_Num(*node*)

render_UnaryOp(*node*)

render_node(*node*)

Functions

brian_ast(expr, variables)	Returns an AST tree representation with additional information

brian_ast function

(*Shortest import*: `from brian2.parsing.bast import brian_ast`)

brian2.parsing.bast.**brian_ast**(*expr*, *variables*)

Returns an AST tree representation with additional information

Each node will be a standard Python `ast` node with the following additional attributes:

dtype One of `'boolean'`, `'integer'` or `'float'`, referring to the data type of the value of this node.

scalar Either `True` or `False` if the node uses any vector-valued variables.

complexity An integer representation of the computational complexity of the node. This is a very rough representation used for things like `2*(x+y)` is less complex than `2*x+2*y` and `exp(x)` is more complex than `2*x` but shouldn't be relied on for fine distinctions between expressions.

Parameters **expr** : str

The expression to convert into an AST representation

variables : dict

The dictionary of `Variable` objects used in the expression.

brian_dtype_from_dtype(dtype)	Returns 'boolean', 'integer' or 'float'

brian_dtype_from_dtype function

(*Shortest import*: `from brian2.parsing.bast import brian_dtype_from_dtype`)

brian2.parsing.bast.**brian_dtype_from_dtype**(*dtype*)

Returns 'boolean', 'integer' or 'float'

brian_dtype_from_value(value)	Returns 'boolean', 'integer' or 'float'

brian_dtype_from_value function

(*Shortest import*: `from brian2.parsing.bast import brian_dtype_from_value`)

brian2.parsing.bast.**brian_dtype_from_value**(*value*)

Returns 'boolean', 'integer' or 'float'

is_boolean(value)	

is_boolean function

(*Shortest import*: `from brian2.parsing.bast import is_boolean`)

brian2.parsing.bast.**is_boolean**(*value*)

is_boolean_dtype(obj)	

is_boolean_dtype function

(*Shortest import*: `from brian2.parsing.bast import is_boolean_dtype`)

brian2.parsing.bast.**is_boolean_dtype**(*obj*)

is_float(value)	

is_float function

(*Shortest import*: `from brian2.parsing.bast import is_float`)

brian2.parsing.bast.**is_float**(*value*)

is_float_dtype(obj)	

is_float_dtype function

(*Shortest import*: `from brian2.parsing.bast import is_float_dtype`)

brian2.parsing.bast.**is_float_dtype**(*obj*)

is_integer(value)	

is_integer function

(*Shortest import*: `from brian2.parsing.bast import is_integer`)

brian2.parsing.bast.**is_integer**(*value*)

is_integer_dtype(obj)	

is_integer_dtype function

(*Shortest import*: `from brian2.parsing.bast import is_integer_dtype`)

brian2.parsing.bast.**is_integer_dtype**(*obj*)

`dependencies` **module**

Exported members: *abstract_code_dependencies*

Functions

abstract_code_dependencies(code[, ...])	Analyses identifiers used in abstract code blocks

abstract_code_dependencies function

(*Shortest import*: `from brian2.parsing.dependencies import abstract_code_dependencies`)

brian2.parsing.dependencies.**abstract_code_dependencies**(*code*, *known_vars=None*, *known_funcs=None*)

Analyses identifiers used in abstract code blocks

> **Parameters** **code** : str
>
>> The abstract code block.
>
>> **known_vars** : set
>
>> The set of known variable names.
>
>> **known_funcs** : set
>
>> The set of known function names.
>
> **Returns** **results** : namedtuple with the following fields
>
>> **all** The set of all identifiers that appear in this code block, including functions.
>
>> **read** The set of values that are read, excluding functions.
>
>> **write** The set of all values that are written to.
>
>> **funcs** The set of all function names.
>
>> **known_all** The set of all identifiers that appear in this code block and are known.
>
>> **known_read** The set of known values that are read, excluding functions.
>
>> **known_write** The set of known values that are written to.
>
>> **known_funcs** The set of known functions that are used.
>
>> **unknown_read** The set of all unknown variables whose values are read. Equal to `read-known_vars`.
>
>> **unknown_write** The set of all unknown variables written to. Equal to `write-known_vars`.
>
>> **unknown_funcs** The set of all unknown function names, equal to `funcs-known_funcs`.

undefined_read The set of all unknown variables whose values are read before they are written to. If this set is nonempty it usually indicates an error, since a variable that is read should either have been defined in the code block (in which case it will appear in `newly_defined`) or already be known.

newly_defined The set of all variable names which are newly defined in this abstract code block.

get_read_write_funcs(parsed_code)	

get_read_write_funcs function

(*Shortest import*: `from brian2.parsing.dependencies import get_read_write_funcs`)

`brian2.parsing.dependencies.`**`get_read_write_funcs`**(*parsed_code*)

expressions module

AST parsing based analysis of expressions

Exported members: *parse_expression_dimensions*

Functions

is_boolean_expression(expr, variables)	Determines if an expression is of boolean type or not

is_boolean_expression function

(*Shortest import*: `from brian2.parsing.expressions import is_boolean_expression`)

`brian2.parsing.expressions.`**`is_boolean_expression`**(*expr*, *variables*)

Determines if an expression is of boolean type or not

> **Parameters** **expr** : str
>
> > The expression to test
>
> **variables** : dict-like of `Variable`
>
> > The variables used in the expression.
>
> **Returns** **isbool** : bool
>
> > Whether or not the expression is boolean.

Raises

SyntaxError If the expression ought to be boolean but is not, for example `x<y and z` where `z` is not a boolean variable.

Notes

We test the following cases recursively on the abstract syntax tree:

- The node is a boolean operation. If all the subnodes are boolean expressions we return `True`, otherwise we raise the `SyntaxError`.
- The node is a function call, we return `True` or `False` depending on whether the function description has the `_returns_bool` attribute set.
- The node is a variable name, we return `True` or `False` depending on whether `is_boolean` attribute is set or if the name is `True` or `False`.
- The node is a comparison, we return `True`.
- The node is a unary operation, we return `True` if the operation is `not`, otherwise `False`.
- Otherwise we return `False`.

parse_expression_dimensions(expr, variables)	Returns the unit value of an expression, and checks its validity

parse_expression_dimensions function

(*Shortest import*: `from brian2.parsing.expressions import parse_expression_dimensions`)

brian2.parsing.expressions.**parse_expression_dimensions**(*expr*, *variables*, *orig_expr=None*)
Returns the unit value of an expression, and checks its validity

> **Parameters** **expr** : str
>
>> The expression to check.
>
> **variables** : dict
>
>> Dictionary of all variables used in the `expr` (including `Constant` objects for external variables)
>
> **Returns** **unit** : Quantity
>
>> The output unit of the expression

Raises

SyntaxError If the expression cannot be parsed, or if it uses `a**b` for `b` anything other than a constant number.

DimensionMismatchError If any part of the expression is dimensionally inconsistent.

`functions` **module**

Exported   members: *AbstractCodeFunction*,   *abstract_code_from_function*,   *extract_abstract_code_functions*, *substitute_abstract_code_functions*

Classes

AbstractCodeFunction(name, args, code, ...)	The information defining an abstract code function

AbstractCodeFunction class

(*Shortest import*: `from brian2.parsing.functions import AbstractCodeFunction`)

class `brian2.parsing.functions.`**AbstractCodeFunction**(*name*, *args*, *code*, *return_expr*)

> Bases: `object`
>
> The information defining an abstract code function
>
> Has attributes corresponding to initialisation parameters
>
> > **Parameters** **name** : str
> >
> > > The function name.
> >
> > **args** : list of str
> >
> > > The arguments to the function.
> >
> > **code** : str
> >
> > > The abstract code string consisting of the body of the function less the return statement.
> >
> > **return_expr** : str or None
> >
> > > The expression returned, or None if there is nothing returned.

FunctionRewriter(func[, numcalls])	Inlines a function call using temporary variables

FunctionRewriter class

(*Shortest import*: `from brian2.parsing.functions import FunctionRewriter`)

class `brian2.parsing.functions.`**FunctionRewriter**(*func*, *numcalls=0*)

> Bases: `ast.NodeTransformer`
>
> Inlines a function call using temporary variables
>
> numcalls is the number of times the function rewriter has been called so far, this is used to make sure that when recursively inlining there is no name aliasing. The substitute_abstract_code_functions ensures that this is kept up to date between recursive runs.
>
> The pre attribute is the set of lines to be inserted above the currently being processed line, i.e. the inline code.
>
> The visit method returns the current line processed so that the function call is replaced with the output of the inlining.

Methods

visit_Call(node)	

Details

visit_Call(*node*)

VarRewriter(pre)	Rewrites all variable names in names by prepending pre

VarRewriter class

(*Shortest import*: `from brian2.parsing.functions import VarRewriter`)

class brian2.parsing.functions.**VarRewriter**(*pre*)
    Bases: `ast.NodeTransformer`

Rewrites all variable names in names by prepending pre

Methods

visit_Call(node)	
visit_Name(node)	

Details

visit_Call(*node*)

visit_Name(*node*)

Functions

abstract_code_from_function(func)	Converts the body of the function to abstract code

abstract_code_from_function function

(*Shortest import*: `from brian2.parsing.functions import abstract_code_from_function`)

brian2.parsing.functions.**abstract_code_from_function**(*func*)
    Converts the body of the function to abstract code

> **Parameters** **func** : function, str or ast.FunctionDef
>
> > The function object to convert. Note that the arguments to the function are ignored.
>
> **Returns** **func** : AbstractCodeFunction

The corresponding abstract code function

Raises

SyntaxError If unsupported features are used such as if statements or indexing.

extract_abstract_code_functions(code)	Returns a set of abstract code functions from function definitions.

extract_abstract_code_functions function

(*Shortest import*: from brian2.parsing.functions import extract_abstract_code_functions)

brian2.parsing.functions.**extract_abstract_code_functions**(*code*)

Returns a set of abstract code functions from function definitions.

Returns all functions defined at the top level and ignores any other code in the string.

Parameters code : str

The code string defining some functions.

Returns funcs : dict

A mapping (name, func) for func an *AbstractCodeFunction*.

substitute_abstract_code_functions(code, funcs)	Performs inline substitution of all the functions in the code

substitute_abstract_code_functions function

(*Shortest import*: from brian2.parsing.functions import substitute_abstract_code_functions)

brian2.parsing.functions.**substitute_abstract_code_functions**(*code*, *funcs*)

Performs inline substitution of all the functions in the code

Parameters code : str

The abstract code to make inline substitutions into.

funcs : list, dict or set of AbstractCodeFunction

The function substitutions to use, note in the case of a dict, the keys are ignored and the function name is used.

Returns code : str

The code with inline substitutions performed.

rendering module

Exported members: [`NodeRenderer`](), [`NumpyNodeRenderer`](), [`CPPNodeRenderer`](), [`SympyNodeRenderer`](), [`get_node_value`]()

Classes

[`CPPNodeRenderer`]([auto_vectorise])	

Methods

CPPNodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import CPPNodeRenderer`)

class brian2.parsing.rendering.**CPPNodeRenderer**(*auto_vectorise=None*)

 Bases: [`brian2.parsing.rendering.NodeRenderer`]()

 Methods

[`render_Assign`](node)	
[`render_BinOp`](node)	
[`render_Name`](node)	
[`render_NameConstant`](node)	

 Details

 render_Assign(*node*)

 render_BinOp(*node*)

 render_Name(*node*)

 render_NameConstant(*node*)

[`NodeRenderer`]([auto_vectorise])	

Methods

NodeRenderer class

(*Shortest import*: `from brian2.parsing.rendering import NodeRenderer`)

class brian2.parsing.rendering.**NodeRenderer**(*auto_vectorise=None*)

   Bases: `object`

Methods

render_Assign(node)	
render_AugAssign(node)	
render_BinOp(node)	
render_BinOp_parentheses(left, right, op)	
render_BoolOp(node)	
render_Call(node)	
render_Compare(node)	
render_Constant(node)	
render_Name(node)	
render_NameConstant(node)	
render_Num(node)	
render_UnaryOp(node)	
render_code(code)	
render_element_parentheses(node)	Render an element with parentheses around it or leave them away for numbers, names and function calls.
render_expr(expr[, strip])	
render_func(node)	
render_node(node)	

Details

render_Assign(*node*)

render_AugAssign(*node*)

render_BinOp(*node*)

render_BinOp_parentheses(*left*, *right*, *op*)

render_BoolOp(*node*)

render_Call(*node*)

render_Compare(*node*)

render_Constant(*node*)

render_Name(*node*)

render_NameConstant(*node*)

render_Num(*node*)

render_UnaryOp(*node*)

render_code(*code*)

render_element_parentheses(*node*)
    Render an element with parentheses around it or leave them away for numbers, names and function calls.

render_expr(*expr*, *strip=True*)

render_func(*node*)

render_node(*node*)

NumpyNodeRenderer([auto_vectorise])

Methods

NumpyNodeRenderer class

(*Shortest import*: from brian2.parsing.rendering import NumpyNodeRenderer)

class brian2.parsing.rendering.**NumpyNodeRenderer**(*auto_vectorise=None*)
    Bases: *brian2.parsing.rendering.NodeRenderer*

Methods

---

render_UnaryOp(node)

Details

render_UnaryOp(*node*)

[*SympyNodeRenderer*](#)([auto_vectorise])

SympyNodeRenderer class

(*Shortest import*: from brian2.parsing.rendering import SympyNodeRenderer)

class brian2.parsing.rendering.**SympyNodeRenderer**(*auto_vectorise=None*)
    Bases: [*brian2.parsing.rendering.NodeRenderer*](#)

Methods

[*render_BinOp*](#)(node)
[*render_BoolOp*](#)(node)
[*render_Call*](#)(node)
[*render_Compare*](#)(node)
[*render_Name*](#)(node)
[*render_NameConstant*](#)(node)
[*render_Num*](#)(node)
[*render_UnaryOp*](#)(node)
[*render_func*](#)(node)

Details

render_BinOp(*node*)

render_BoolOp(*node*)

render_Call(*node*)

render_Compare(*node*)

render_Name(*node*)

render_NameConstant(*node*)

render_Num(*node*)

---

render_UnaryOp(*node*)

render_func(*node*)

Functions

get_node_value(node)	Helper function to mask differences between Python versions

get_node_value function

(*Shortest import*: `from brian2.parsing.rendering import get_node_value`)

`brian2.parsing.rendering.`**`get_node_value`**(*node*)

Helper function to mask differences between Python versions

statements module

Functions

parse_statement(code)	Parses a single line of code into "var op expr".

parse_statement function

(*Shortest import*: `from brian2.parsing.statements import parse_statement`)

`brian2.parsing.statements.`**`parse_statement`**(*code*)

Parses a single line of code into "var op expr".

> **Parameters** **code** : str
>
> > A string containing a single statement of the form `var op expr # comment`, where the `# comment` part is optional.
>
> **Returns** **var, op, expr, comment** : str, str, str, str
>
> > The four parts of the statement.

Examples

```
>>> parse_statement('v = -65*mV  # reset the membrane potential')
('v', '=', '-65*mV', 'reset the membrane potential')
>>> parse_statement('v += dt*(-v/tau)')
('v', '+=', 'dt*(-v/tau)', '')
```

`sympytools` **module**

Utility functions for parsing expressions and statements.

Classes

[*CustomSympyPrinter*](#)([settings])	Printer that overrides the printing of some basic sympy objects.

CustomSympyPrinter class

(*Shortest import*: `from brian2.parsing.sympytools import CustomSympyPrinter`)

class brian2.parsing.sympytools.**CustomSympyPrinter**(*settings=None*)
Bases: [`sympy.printing.str.StrPrinter`](#)

Printer that overrides the printing of some basic sympy objects. E.g. print "a and b" instead of "And(a, b)".

Functions

[*check_expression_for_multiple_stateful_functions*](#)(...)	

check_expression_for_multiple_stateful_functions function

(*Shortest import*: `from brian2.parsing.sympytools import check_expression_for_multiple_stateful_func`

brian2.parsing.sympytools.**check_expression_for_multiple_stateful_functions**(*expr*, *variables*)

[*expression_complexity*](#)(expr[, complexity])	Returns the complexity of an expression (either string or sympy)

expression_complexity function

(*Shortest import*: `from brian2.parsing.sympytools import expression_complexity`)

brian2.parsing.sympytools.**expression_complexity**(*expr*, *complexity=None*)
Returns the complexity of an expression (either string or sympy)

The complexity is defined as 1 for each arithmetic operation except divide which is 2, and all other operations are 20. This can be overridden using the complexity argument.

Note: calling this on a statement rather than an expression is likely to lead to errors.

> **Parameters** **expr: `sympy.Expr` or str** :
>
> > The expression.
>
> **complexity: None or dict (optional)** :
>
> > A dictionary mapping expression names to their complexity, to overwrite default behaviour.

Returns complexity: int :

> The complexity of the expression.

str_to_sympy(expr[, variables])	Parses a string into a sympy expression.

str_to_sympy function

(*Shortest import*: `from brian2.parsing.sympytools import str_to_sympy`)

`brian2.parsing.sympytools.`**`str_to_sympy`**(*expr*, *variables=None*)

> Parses a string into a sympy expression. There are two reasons for not using `sympify` directly: 1) sympify does a `from sympy import *`, adding all functions to its namespace. This leads to issues when trying to use sympy function names as variable names. For example, both `beta` and `factor` – quite reasonable names for variables – are sympy functions, using them as variables would lead to a parsing error. 2) We want to use a common syntax across expressions and statements, e.g. we want to allow to use `and` (instead of `&`) and function names like `ceil` (instead of `ceiling`).

> **Parameters expr** : str
>
> > The string expression to parse.
>
> **variables** : dict, optional
>
> > Dictionary mapping variable/function names in the expr to their respective `Variable`/*Function* objects.
>
> **Returns s_expr** :
>
> > A sympy expression

Raises

SyntaxError In case of any problems during parsing.

sympy_to_str(sympy_expr)	Converts a sympy expression into a string.

sympy_to_str function

(*Shortest import*: `from brian2.parsing.sympytools import sympy_to_str`)

`brian2.parsing.sympytools.`**`sympy_to_str`**(*sympy_expr*)

> Converts a sympy expression into a string. This could be as easy as `str(sympy_exp)` but it is possible that the sympy expression contains functions like `Abs` (for example, if an expression such as `sqrt(x**2)` appeared somewhere). We do want to re-translate `Abs` into `abs` in this case.

> **Parameters sympy_expr** : sympy.core.expr.Expr
>
> > The expression that should be converted to a string.
>
> **Returns** :
>
> **str_expr** : str
>
> > A string representing the sympy expression.

Objects

PRINTER	Printer that overrides the printing of some basic sympy objects.

PRINTER object

(*Shortest import*: `from brian2.parsing.sympytools import PRINTER`)

`brian2.parsing.sympytools.`**`PRINTER =`**
`<brian2.parsing.sympytools.CustomSympyPrinter object>`
   Printer that overrides the printing of some basic sympy objects. E.g. print "a and b" instead of "And(a, b)".

6.7.11 random package

6.7.12 spatialneuron package

Exported members: *Morphology*, *Soma*, *Cylinder*, *Section*, *SpatialNeuron*

`morphology` module

Neuronal morphology module. This module defines classes to load and build neuronal morphologies.

Exported members: *Morphology*, *Section*, *Cylinder*, *Soma*

Classes

Children(owner)	Helper class to represent the children (sub trees) of a section.

Children class

(*Shortest import*: `from brian2.spatialneuron.morphology import Children`)

class `brian2.spatialneuron.morphology.`**`Children`**(*owner*)
   Bases: `object`

   Helper class to represent the children (sub trees) of a section. Can be used like a dictionary (mapping names to *Morphology* objects), but iterates over the values (sub trees) instead of over the keys (names).

   #### Methods

add(name, subtree[, automatic_name])	Add a new child to the morphology.
name(child)	Return the given name (i.e.
remove(name)	Remove a subtree from this morphology.

Details

add (*name*, *subtree*, *automatic_name=False*)

Add a new child to the morphology.

> **Parameters** **name** : str
>
> > The name (e.g. `"axon"`, `"soma"`) to use for this sub tree.
>
> **subtree** : *Morphology*
>
> > The subtree to link as a child.
>
> **automatic_name** : bool, optional
>
> > Whether to chose a new name automatically, if a subtree of the same name already exists
> > (uses e.g. `"dend2"` instead `"dend"`). Defaults to `False` and will raise an error instead.

name (*child*)

Return the given name (i.e. not the automatic name such as `1`) for a child subtree.

> **Parameters** **child** : *Morphology*
>
> **Returns** **name** : str
>
> > The given name for the `child`.

remove (*name*)

Remove a subtree from this morphology.

> **Parameters** **name** : str
>
> > The name of the sub tree to remove.

Cylinder(**kwds)	A cylindrical section.

Cylinder class

(*Shortest import*: `from brian2 import Cylinder`)

class brian2.spatialneuron.morphology.**Cylinder**(***kwds*)

> Bases: *brian2.spatialneuron.morphology.Section*

A cylindrical section. For sections with more complex geometry (varying length and/or diameter of each compartment), use the *Section* class.

> **Parameters** **diameter** : *Quantity*
>
> > The diameter of the cylinder.
>
> **n** : int, optional
>
> > The number of compartments in this section. Defaults to 1.
>
> **length** : *Quantity*, optional
>
> > The length of the cylinder. Cannot be combined with the specification of coordinates.
>
> **x** : *Quantity*, optional
>
> > A sequence of two values, the start and the end point of the cylinder. The coordinates
> > are interpreted as relative to the end point of the parent compartment (if any), so in most
> > cases the start point should be `0*um`. The common exception is a cylinder connecting to
> > a *Soma*, here the start point can be used to make the cylinder start at the surface of the

sphere instead of at its center. You can specify all of x, y, or z to specify a morphology in 3D, or only one or two out of them to specify a morphology in 1D or 2D.

y : *Quantity*, optional

> See x

z : *Quantity*, optional

> See x

type : str, optional

> The type (e.g. "axon") of this *Cylinder*.

Attributes

area	The membrane surface area of each compartment in this section.
diameter	The diameter at the middle of each compartment in this section.
end_diameter	The diameter at the end of each compartment in this section.
r_length_1	The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment.
r_length_2	The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment.
start_diameter	The diameter at the start of each compartment in this section.
volume	The volume of each compartment in this section.

Methods

copy_section()	Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

Details

area

> The membrane surface area of each compartment in this section. The surface area of each compartment is calculated as $\pi d l$, where l is the length of the compartment, and d is its diameter. Note that this surface area does not contain the area of the two disks at the two sides of the cylinder.

diameter

> The diameter at the middle of each compartment in this section.

end_diameter

> The diameter at the end of each compartment in this section.

r_length_1

> The geometry-dependent term to calculate the conductance between the start and the midpoint of each com-

partment. Dividing this value by the Intracellular resistivity gives the conductance.

r_length_2
> The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment. Dividing this value by the Intracellular resistivity gives the conductance.

start_diameter
> The diameter at the start of each compartment in this section.

volume

> The volume of each compartment in this section. The volume of each compartment is calculated as $\pi \frac{d}{2}^2 l$, where l is the length of the compartment, and d is its diameter.

copy_section()
> Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

> > **Returns** **copy** : *Morphology*

> > A copy of this section (without the links to the parent/children)

Tutorials and examples using this

- Example *compartmental/bipolar_cell*
- Example *compartmental/bipolar_with_inputs*
- Example *compartmental/bipolar_with_inputs2*
- Example *compartmental/cylinder*
- Example *compartmental/hh_with_spikes*
- Example *compartmental/hodgkin_huxley_1952*
- Example *compartmental/infinite_cable*
- Example *compartmental/lfp*
- Example *compartmental/morphotest*
- Example *compartmental/rall*
- Example *compartmental/spike_initiation*
- Example *frompapers/Brette_2012/Fig1*
- Example *frompapers/Brette_2012/Fig3AB*
- Example *frompapers/Brette_2012/Fig3CF*
- Example *frompapers/Brette_2012/Fig5A*
- Example *frompapers/Destexhe_et_al_1998*

Morphology(**kwds)	Neuronal morphology (tree structure).

Morphology class

(*Shortest import*: `from brian2 import Morphology`)

class `brian2.spatialneuron.morphology.`**`Morphology`**(***kwds*)
 Bases: [`object`](#)

 Neuronal morphology (tree structure).

 The data structure is a tree where each node is an un-branched section consisting of a number of connected compartments, each one defined by its geometrical properties (length, area, diameter, position).

Notes

 You cannot create objects of this class, create a [`Soma`](#), a [`Section`](#), or a [`Cylinder`](#) instead.

Attributes

[*area*](#)	The membrane surface area of each compartment in this section.
[*children*](#)	The children (as a [`Children`](#) object) of this section.
[*coordinates*](#)	Array with all coordinates at the start- and end-points of each compartment in this section.
[*coordinates_*](#)	Array with all coordinates (as unitless floating point numbers) at the start- and end-points of each compartment in this section.
[*diameter*](#)	The diameter at the middle of each compartment in this section.
[*distance*](#)	The total distance between the midpoint of each compartment and the root of the morphology.
[*end_distance*](#)	The distance to the root of the morphology at the end of this section.
[*end_x*](#)	The x coordinate at the end of each compartment.
[*end_x_*](#)	The x coordinate (as a unitless floating point number) at the end of each compartment.
[*end_y*](#)	The y coordinate at the end of each compartment.
[*end_y_*](#)	The y coordinate (as a unitless floating point number) at the end of each compartment.
[*end_z*](#)	The z coordinate at the end of each compartment.
[*end_z_*](#)	The z coordinate (as a unitless floating point number) at the end of each compartment.
[*length*](#)	The length of each compartment in this section.
[*n*](#)	The number of compartments in this section.
[*parent*](#)	The parent section of this section.
[*r_length_1*](#)	The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment.
[*r_length_2*](#)	The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment.

Table 379 – continued from previous page

start_x	The x coordinate at the beginning of each compartment.
*start_x_*	The x coordinate (as a unitless floating point number) at the beginning of each compartment.
start_y	The y coordinate at the beginning of each compartment.
*start_y_*	The y coordinate (as a unitless floating point number) at the beginning of each compartment.
start_z	The z coordinate at the beginning of each compartment.
*start_z_*	The z coordinate (as a unitless floating point number) at the beginning of each compartment.
total_compartments	The total number of compartments in this subtree (i.e.
total_sections	The total number of sections in this subtree.
volume	The volume of each compartment in this section.
x	The x coordinate at the midpoint of each compartment.
*x_*	The x coordinate (as a unitless floating point number) at the midpoint of each compartment.
y	The y coordinate at the midpoint of each compartment.
*y_*	The y coordinate (as a unitless floating point number) at the midpoint of each compartment.
z	The y coordinate at the midpoint of each compartment.
*z_*	The z coordinate (as a unitless floating point number) at the midpoint of each compartment.

Methods

copy_section()	Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)
from_file(filename[, spherical_soma])	Convencience method to load a morphology from a given file.
from_points(points[, spherical_soma])	Create a morphology from a sequence of points (similar to the SWC format, see *Morphology.from_swc_file*).
from_swc_file(filename[, spherical_soma])	Load a morphology from a SWC file.
generate_coordinates([section_randomness, ...])	Create a new *Morphology*, with coordinates filled in place where the previous morphology did not have any.
topology()	Return a representation of the topology

Details

area
>   The membrane surface area of each compartment in this section.

children
>   The children (as a *Children* object) of this section.

coordinates
>   Array with all coordinates at the start- and end-points of each compartment in this section. The array has size $(n + 1) \times 3$, where n is the number of compartments in this section. Each row is one point (start point of first compartment, end point of first compartment, end point of second compartment, …), with the columns being the x, y, and z coordinates. Returns `None` for morphologies without coordinates.

coordinates_
>   Array with all coordinates (as unitless floating point numbers) at the start- and end-points of each compartment in this section. The array has size $(n + 1) \times 3$, where n is the number of compartments in this section. Each row is one point (start point of first compartment, end point of first compartment, end point of second compartment, …), with the columns being the x, y, and z coordinates. Returns `None` for morphologies without coordinates.

diameter
>   The diameter at the middle of each compartment in this section.

distance
>   The total distance between the midpoint of each compartment and the root of the morphology.

end_distance
>   The distance to the root of the morphology at the end of this section.

end_x
>   The x coordinate at the end of each compartment. Returns `None` for morphologies without coordinates.

end_x_
>   The x coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

end_y
>   The y coordinate at the end of each compartment. Returns `None` for morphologies without coordinates.

end_y_
>   The y coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

end_z
>   The z coordinate at the end of each compartment. Returns `None` for morphologies without coordinates.

end_z_
>   The z coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

length
>   The length of each compartment in this section.

n
>   The number of compartments in this section.

parent
>   The parent section of this section.

r_length_1
> The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment. Dividing this value by the Intracellular resistivity gives the conductance.

r_length_2
> The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment. Dividing this value by the Intracellular resistivity gives the conductance.

start_x
> The x coordinate at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_x_
> The x coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_y
> The y coordinate at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_y_
> The y coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_z
> The z coordinate at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_z_
> The z coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

total_compartments
> The total number of compartments in this subtree (i.e. the number of compartments in this section plus all the compartments in the sections deeper in the tree).

total_sections
> The total number of sections in this subtree.

volume
> The volume of each compartment in this section.

x
> The x coordinate at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

x_
> The x coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

y
> The y coordinate at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

y_
> The y coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

z
> The y coordinate at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

z_
> The z coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

abstract copy_section()
> Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

Returns copy : *Morphology*

A copy of this section (without the links to the parent/children)

static from_file(*filename*, *spherical_soma=True*)

Convencience method to load a morphology from a given file. At the moment, only SWC files are supported, calling this function is therefore equivalent to calling *Morphology.from_swc_file* directly.

Parameters filename : str

The name of a file storing a morphology.

spherical_soma : bool, optional

Whether to model the soma as a sphere.

Returns :

——- :

morphology : *Morphology*

The morphology stored in the given file.

static from_points(*points*, *spherical_soma=True*)

Create a morphology from a sequence of points (similar to the SWC format, see *Morphology. from_swc_file*). Each point has to be a 7-tuple: (index, name, x, y, z, diameter, parent)

Note that the values should not use units, but are instead all taken to be in micrometers.

Parameters points : sequence of 7-tuples

The points of the morphology.

spherical_soma : bool, optional

Whether to model a soma as a sphere.

Returns :

——- :

morphology : *Morphology*

Notes

This format closely follows the SWC format (see *Morphology.from_swc_file*) with two differences: the type should be a string (e.g. 'soma') instead of an integer and the 6-th element should be the diameter and not the radius.

static from_swc_file(*filename*, *spherical_soma=True*)

Load a morphology from a SWC file. A large database of morphologies in this format can be found at http://neuromorpho.org

The format consists of an optional header of lines starting with # (ignored), followed by a sequence of points, each described in a line following the format:

```
index type x y z radius parent
```

index is an integer label (starting at 1) that identifies the current point and increases by one each line. type is an integer representing the type of the neural segment. The only type that changes the interpretation by Brian is the type 1 which signals a soma. Types 2 (axon), 3 (dendrite), and 4 (apical dendrite) are used to give corresponding names to the respective sections. All other types are ignored. x, y, and z are the cartesian

coordinates at each point and `r` is its radius. `parent` refers to the index of the parent point or is `-1` for the root point.

> **Parameters filename** : str
>
> > The name of the `SWC` file.
>
> > **spherical_soma** : bool, optional
> >
> > Whether to model the soma as a sphere.
>
> **Returns morpho** : *Morphology*
>
> > The morphology stored in the given file.

generate_coordinates(*section_randomness=0.0*, *compartment_randomness=0.0*, *overwrite_existing=False*)

Create a new *Morphology*, with coordinates filled in place where the previous morphology did not have any. This is mostly useful for plotting a morphology, it does not affect its electrical properties.

> **Parameters section_randomness** : float, optional
>
> > The randomness when deciding the direction vector for each new section. The given number is the β parameter of an exponential distribution (in degrees) which will be used to determine the deviation from the direction of the parent section. If the given value equals 0 (the default), then a deterministic algorithm will be used instead.
>
> > **compartment_randomness** : float, optional
> >
> > The randomness when deciding the direction vector for each compartment within a section. The given number is the β parameter of an exponential distribution (in degrees) which will be used to determine the deviation from the main direction of the current section. If the given value equals 0 (the default), then all compartments will be along a straight line.
>
> > **overwrite_existing** : bool, optional
> >
> > Whether to overwrite existing coordinates in the morphology. This is by default set to `False`, meaning that only sections that do not currently have any coordinates set will get new coordinates. This allows to conveniently generate a morphology that can be plotted for a morphology that is based on points but also has artificially added sections (the most common case: an axon added to a reconstructed morphology). If set to `True`, all sections will get new coordinates. This can be useful to either get a schematic representation of the morphology (with `section_randomness` and `compartment_randomness` both 0) or to simply generate a new random variation of a morphology (which will still be electrically equivalent, of course).
>
> **Returns morpho_with_coordinates** : *Morphology*
>
> > The same morphology, but with coordinates

topology()

Return a representation of the topology

> **Returns topology** : *Topology*
>
> > An object representing the topology (can be converted to a string by using `str(...)` or simply by printing it with `print`.)

Tutorials and examples using this

- Example *compartmental/bipolar_cell*
- Example *compartmental/bipolar_with_inputs*
- Example *compartmental/bipolar_with_inputs2*
- Example *compartmental/cylinder*
- Example *compartmental/infinite_cable*
- Example *compartmental/morphotest*
- Example *compartmental/rall*
- Example *compartmental/spike_initiation*
- Example *frompapers/Brette_2012/Fig1*
- Example *frompapers/Brette_2012/Fig3AB*
- Example *frompapers/Brette_2012/Fig3CF*
- Example *frompapers/Brette_2012/Fig4*
- Example *frompapers/Brette_2012/Fig5A*

`MorphologyIndexWrapper`(morphology)	A simpler version of `IndexWrapper`, not allowing for string indexing (`Morphology` is not a `Group`).

MorphologyIndexWrapper class

(*Shortest import*: `from brian2.spatialneuron.morphology import MorphologyIndexWrapper`)

class brian2.spatialneuron.morphology.**MorphologyIndexWrapper**(*morphology*)
 Bases: `object`

 A simpler version of `IndexWrapper`, not allowing for string indexing (`Morphology` is not a `Group`). It allows to use `morphology.indices[...]` instead of `morphology[...]._indices()`.

`Node`(index, comp_name, x, y, z, diameter, ...)	

Attributes

Node class

(*Shortest import*: `from brian2.spatialneuron.morphology import Node`)

class `brian2.spatialneuron.morphology.`**Node**(*index*, *comp_name*, *x*, *y*, *z*, *diameter*, *parent*, *children*)
    Bases: `tuple`

Attributes

children	Alias for field number 7
comp_name	Alias for field number 1
diameter	Alias for field number 5
index	Alias for field number 0
parent	Alias for field number 6
x	Alias for field number 2
y	Alias for field number 3
z	Alias for field number 4

Details

children
    Alias for field number 7

comp_name
    Alias for field number 1

diameter
    Alias for field number 5

index
    Alias for field number 0

parent
    Alias for field number 6

x
    Alias for field number 2

y
    Alias for field number 3

z
    Alias for field number 4

Section(**kwds)	A section (unbranched structure), described as a sequence of truncated cones with potentially varying diameters and lengths per compartment.

Section class

(*Shortest import*: `from brian2 import Section`)

class `brian2.spatialneuron.morphology.`**`Section`**`(**kwds)`
Bases: *`brian2.spatialneuron.morphology.Morphology`*

A section (unbranched structure), described as a sequence of truncated cones with potentially varying diameters and lengths per compartment.

> **Parameters** **diameter** : *`Quantity`*
>
>> Either a single value (the constant diameter along the whole section), or a value of length `n+1`. When `n+1` values are given, they will be interpreted as the diameters at the start of the first compartment and the diameters at the end of each compartment (which is equivalent to: the diameter at the start of each compartment and the diameter at the end of the last compartment.
>
> **n** : int, optional
>
>> The number of compartments in this section. Defaults to 1.
>
> **length** : *`Quantity`*, optional
>
>> Either a single value (the total length of the section), or a value of length `n`, the length of each individual compartment. Cannot be combined with the specification of coordinates.
>
> **x** : *`Quantity`*, optional
>
>> `n+1` values, specifying the x coordinates of the start point of the first compartment and the end-points of all compartments (which is equivalent to: the start point of all compartments and the end point of the last compartment). The coordinates are interpreted as relative to the end point of the parent compartment (if any), so in most cases the start point should be `0*um`. The common exception is a cylinder connecting to a *`Soma`*, here the start point can be used to make the cylinder start at the surface of the sphere instead of at its center. You can specify all of x, y, or z to specify a morphology in 3D, or only one or two out of them to specify a morphology in 1D or 2D.
>
> **y** : *`Quantity`*, optional
>
>> See x
>
> **z** : *`Quantity`*, optional
>
>> See x
>
> **type** : str, optional
>
>> The type (e.g. `"axon"`) of this *`Section`*.

Attributes

area	The membrane surface area of each compartment in this section.
diameter	The diameter at the middle of each compartment in this section.
distance	The total distance between the midpoint of each compartment and the root of the morphology.

end_diameter	The diameter at the end of each compartment in this section.
end_distance	The distance to the root of the morphology at the end of this section.
*end_x_*	The x coordinate (as a unitless floating point number) at the end of each compartment.
*end_y_*	The y coordinate (as a unitless floating point number) at the end of each compartment.
*end_z_*	The z coordinate (as a unitless floating point number) at the end of each compartment.
length	The length of each compartment in this section.
r_length_1	The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment.
r_length_2	The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment.
start_diameter	The diameter at the start of each compartment in this section.
*start_x_*	The x coordinate (as a unitless floating point number) at the beginning of each compartment.
*start_y_*	The y coordinate (as a unitless floating point number) at the beginning of each compartment.
*start_z_*	The z coordinate (as a unitless floating point number) at the beginning of each compartment.
volume	The volume of each compartment in this section.
*x_*	The x coordinate (as a unitless floating point number) at the midpoint of each compartment.
*y_*	The y coordinate (as a unitless floating point number) at the midpoint of each compartment.
*z_*	The z coordinate (as a unitless floating point number) at the midpoint of each compartment.

Methods

copy_section()	Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

Details

area

The membrane surface area of each compartment in this section. The surface area of each compartment is calculated as $\frac{\pi}{2}(d_1 + d_2)\sqrt{\frac{(d_1-d_2)^2}{4} + l^2}$, where l is the length of the compartment, and d_1 and d_2 are the diameter at the start and end of the compartment, respectively. Note that this surface area does not contain the area of the two disks at the two sides of the truncated cone.

diameter

The diameter at the middle of each compartment in this section.

distance

The total distance between the midpoint of each compartment and the root of the morphology.

end_diameter

The diameter at the end of each compartment in this section.

end_distance

The distance to the root of the morphology at the end of this section.

end_x_

The x coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

end_y_

The y coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

end_z_

The z coordinate (as a unitless floating point number) at the end of each compartment. Returns `None` for morphologies without coordinates.

length

The length of each compartment in this section.

r_length_1

The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment. Dividing this value by the Intracellular resistivity gives the conductance.

r_length_2

The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment. Dividing this value by the Intracellular resistivity gives the conductance.

start_diameter

The diameter at the start of each compartment in this section.

start_x_

The x coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_y_

The y coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

start_z_

The z coordinate (as a unitless floating point number) at the beginning of each compartment. Returns `None` for morphologies without coordinates.

volume

The volume of each compartment in this section. The volume of each compartment is calculated as $\frac{\pi}{12}l(d_1^2 + d_1 d_2 + d_2^2)$, where l is the length of the compartment, and d_1 and d_2 are the diameter at the start and end of the compartment, respectively.

x_

The x coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

y_

The y coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

z_
> The z coordinate (as a unitless floating point number) at the midpoint of each compartment. Returns `None` for morphologies without coordinates.

copy_section()
> Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

> > **Returns copy** : *Morphology*

> > A copy of this section (without the links to the parent/children)

Tutorials and examples using this

- Example *frompapers/Brette_2012/Fig4*

Soma(**kwds)	A spherical, iso-potential soma.

Soma class

(*Shortest import*: `from brian2 import Soma`)

class brian2.spatialneuron.morphology.**Soma**(***kwds*)
> Bases: *brian2.spatialneuron.morphology.Morphology*

> A spherical, iso-potential soma.

> > **Parameters diameter** : *Quantity*

> > > Diameter of the sphere.

> > **x** : *Quantity*, optional

> > > The x coordinate of the position of the soma.

> > **y** : *Quantity*, optional

> > > The y coordinate of the position of the soma.

> > **z** : *Quantity*, optional

> > > The z coordinate of the position of the soma.

> > **type** : str, optional

> > > The `type` of this section, defaults to `'soma'`.

Attributes

area	The membrane surface area of this section (as an array of length 1).
diameter	The diameter of this section (as an array of length 1).
distance	The total distance between the midpoint of this section and the root of the morphology.
end_distance	The distance to the root of the morphology at the end of this section.

continues on next page

Table 388 – continued from previous page

*end_x_*	The x-coordinate of the current section (as an array of length 1).
*end_y_*	The y-coordinate of the current section (as an array of length 1).
*end_z_*	The z-coordinate of the current section (as an array of length 1).
length	The "length" (equal to *diameter*) of this section (as an array of length 1).
r_length_1	The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment.
r_length_2	The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment.
*start_x_*	The x-coordinate of the current section (as an array of length 1).
*start_y_*	The y-coordinate of the current section (as an array of length 1).
*start_z_*	The z-coordinate of the current section (as an array of length 1).
volume	The volume of this section (as an array of length 1).
*x_*	The x-coordinate of the current section (as an array of length 1).
*y_*	The y-coordinate of the current section (as an array of length 1).
*z_*	The z-coordinate of the current section (as an array of length 1).

Methods

copy_section()	Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

Details

area
  The membrane surface area of this section (as an array of length 1).

diameter
  The diameter of this section (as an array of length 1).

distance
  The total distance between the midpoint of this section and the root of the morphology. The *Soma* is most likely the root of the morphology, and therefore the *distance* is 0.

end_distance
  The distance to the root of the morphology at the end of this section. Note that since a *Soma* is modeled as a point (see docs of x, etc.), it does not add anything to the total distance, e.g. a section connecting to a *Soma* has a *distance* of 0 um at its start.

end_x_
> The x-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

end_y_
> The y-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

end_z_
> The z-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

length
> The "length" (equal to *diameter*) of this section (as an array of length 1).

r_length_1
> The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment. Returns a fixed (high) value for a *Soma*, corresponding to a section with very low intracellular resistance.

r_length_2
> The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment. Returns a fixed (high) value for a *Soma*, corresponding to a section with very low intracellular resistance.

start_x_
> The x-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

start_y_
> The y-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

start_z_
> The z-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

volume
> The volume of this section (as an array of length 1).

x_
> The x-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

y_
> The y-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

z_
> The z-coordinate of the current section (as an array of length 1). Note that a *Soma* is modelled as a "point" with finite surface/volume, equivalent to that of a sphere with the given *diameter*. It's start-, midpoint-, and end-coordinates are therefore identical.

copy_section()

    Create a copy of the current section (attributes of this section only, not re-creating the parent/children relation)

        Returns copy : *Morphology*

           A copy of this section (without the links to the parent/children)

Tutorials and examples using this

- Example *compartmental/bipolar_cell*

- Example *compartmental/bipolar_with_inputs*

- Example *compartmental/bipolar_with_inputs2*

- Example *compartmental/morphotest*

- Example *compartmental/spike_initiation*

- Example *frompapers/Brette_2012/Fig1*

- Example *frompapers/Brette_2012/Fig3AB*

- Example *frompapers/Brette_2012/Fig3CF*

- Example *frompapers/Brette_2012/Fig4*

- Example *frompapers/Brette_2012/Fig5A*

SubMorphology(morphology, i, j)	A view on a subset of a section in a morphology.

SubMorphology class

(*Shortest import*: `from brian2.spatialneuron.morphology import SubMorphology`)

class `brian2.spatialneuron.morphology.`**SubMorphology**(*morphology*, *i*, *j*)

    Bases: `object`

    A view on a subset of a section in a morphology.

Attributes

area	The membrane surface area of each compartment in this sub-section.
diameter	The diameter at the middle of each compartment in this sub-section.
distance	The total distance between the midpoint of each compartment in this sub-section and the root of the morphology.
end_x	The x coordinate at the end of each compartment in this sub-section.
*end_x_*	The x coordinate (as a unitless floating point number) at the end of each compartment in this sub-section.
end_y	The y coordinate at the end of each compartment in this sub-section.

continues on next page

Table 391 – continued from previous page

*end_y_*	The y coordinate (as a unitless floating point number) at the end of each compartment in this sub-section.
end_z	The z coordinate at the end of each compartment in this sub-section.
*end_z_*	The z coordinate (as a unitless floating point number) at the end of each compartment in this sub-section.
length	The length of each compartment in this sub-section.
n	The number of compartments in this sub-section.
n_sections	The number of sections in this sub-section (always 1).
r_length_1	The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment in this sub-section.
r_length_2	The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment in this sub-section.
start_x	The x coordinate at the beginning of each compartment in this sub-section.
*start_x_*	The x coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section.
start_y	The y coordinate at the beginning of each compartment in this sub-section.
*start_y_*	The y coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section.
start_z	The x coordinate at the beginning of each compartment in this sub-section.
*start_z_*	The z coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section.
volume	The volume of each compartment in this sub-section.
x	The x coordinate at the midpoint of each compartment in this sub-section.
*x_*	The x coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section.
y	The y coordinate at the midpoint of each compartment in this sub-section.
*y_*	The y coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section.
z	The z coordinate at the midpoint of each compartment in this sub-section.
*z_*	The z coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section.

Details

area
>   The membrane surface area of each compartment in this sub-section.

diameter
>   The diameter at the middle of each compartment in this sub-section.

distance
>   The total distance between the midpoint of each compartment in this sub-section and the root of the morphology.

end_x
>   The x coordinate at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

end_x_
>   The x coordinate (as a unitless floating point number) at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

end_y
>   The y coordinate at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

end_y_
>   The y coordinate (as a unitless floating point number) at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

end_z
>   The z coordinate at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

end_z_
>   The z coordinate (as a unitless floating point number) at the end of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

length
>   The length of each compartment in this sub-section.

n
>   The number of compartments in this sub-section.

n_sections
>   The number of sections in this sub-section (always 1).

r_length_1
>   The geometry-dependent term to calculate the conductance between the start and the midpoint of each compartment in this sub-section. Dividing this value by the Intracellular resistivity gives the conductance.

r_length_2
>   The geometry-dependent term to calculate the conductance between the midpoint and the end of each compartment in this sub-section. Dividing this value by the Intracellular resistivity gives the conductance.

start_x
>   The x coordinate at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

start_x_
>   The x coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

start_y
> The y coordinate at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

start_y_
> The y coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

start_z
> The x coordinate at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

start_z_
> The z coordinate (as a unitless floating point number) at the beginning of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

volume
> The volume of each compartment in this sub-section.

x
> The x coordinate at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

x_
> The x coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

y
> The y coordinate at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

y_
> The y coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

z
> The z coordinate at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

z_
> The z coordinate (as a unitless floating point number) at the midpoint of each compartment in this sub-section. Returns `None` for morphologies without coordinates.

Topology(morphology)	A representation of the topology of a *Morphology*.

Topology class

(*Shortest import*: `from brian2.spatialneuron.morphology import Topology`)

class brian2.spatialneuron.morphology.**Topology**(*morphology*)
> Bases: `object`

> A representation of the topology of a *Morphology*. Has a useful string representation, inspired by NEURON's `topology` function.

spatialneuron module

Compartmental models. This module defines the *SpatialNeuron* class, which defines multicompartmental models.

Exported members: *SpatialNeuron*

Classes

FlatMorphology(morphology)	Container object to store the flattened representation of a morphology.

FlatMorphology class

(*Shortest import*: `from brian2.spatialneuron.spatialneuron import FlatMorphology`)

class `brian2.spatialneuron.spatialneuron.`**FlatMorphology**(*morphology*)
Bases: `object`

Container object to store the flattened representation of a morphology. Note that all values are stored as numpy arrays without unit information (i.e. in base units).

SpatialNeuron(*args, **kw)	A single neuron with a morphology and possibly many compartments.

SpatialNeuron class

(*Shortest import*: `from brian2 import SpatialNeuron`)

class `brian2.spatialneuron.spatialneuron.`**SpatialNeuron**(**args*, ***kw*)
Bases: *brian2.groups.neurongroup.NeuronGroup*

A single neuron with a morphology and possibly many compartments.

> **Parameters** **morphology** : *Morphology*
>
> > The morphology of the neuron.
>
> **model** : str, *Equations*
>
> > The equations defining the group.
>
> **method** : str, function, optional
>
> > The numerical integration method. Either a string with the name of a registered method (e.g. "euler") or a function that receives an *Equations* object and returns the corresponding abstract code. If no method is specified, a suitable method will be chosen automatically.
>
> **threshold** : str, optional
>
> > The condition which produces spikes. Should be a single line boolean expression.
>
> **threshold_location** : (int, *Morphology*), optional
>
> > Compartment where the threshold condition applies, specified as an integer (compartment index) or a *Morphology* object corresponding to the compartment (e.g. `morpho.axon[10*um]`). If unspecified, the threshold condition applies at all compartments.
>
> **Cm** : *Quantity*, optional

Specific capacitance in uF/cm**2 (default 0.9). It can be accessed and modified later as a state variable. In particular, its value can differ in different compartments.

Ri : *Quantity*, optional

Intracellular resistivity in ohm.cm (default 150). It can be accessed as a shared state variable, but modified only before the first run. It is uniform across the neuron.

reset : str, optional

The (possibly multi-line) string with the code to execute on reset.

events : dict, optional

User-defined events in addition to the "spike" event defined by the `threshold`. Has to be a mapping of strings (the event name) to strings (the condition) that will be checked.

refractory : {str, *Quantity*}, optional

Either the length of the refractory period (e.g. `2*ms`), a string expression that evaluates to the length of the refractory period after each spike (e.g. `'(1 + rand())*ms'`), or a string expression evaluating to a boolean value, given the condition under which the neuron stays refractory after a spike (e.g. `'v > -20*mV'`)

namespace : dict, optional

A dictionary mapping identifier names to objects. If not given, the namespace will be filled in at the time of the call of *Network.run*, with either the values from the `namespace` argument of the *Network.run* method or from the local context, if no such argument is given.

dtype : (`dtype`, `dict`), optional

The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

dt : *Quantity*, optional

The time step to be used for the simulation. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

name : str, optional

A unique name for the group, otherwise use `spatialneuron_0`, etc.

Attributes

user_equations	The original equations as specified by the user (i.e.

Methods

spatialneuron_attribute(neuron, name)	Selects a subtree from *SpatialNeuron* neuron and returns a *SpatialSubgroup*.
spatialneuron_segment(neuron, item)	Selects a segment from *SpatialNeuron* neuron, where item is a slice of either compartment indexes or distances.

Details

user_equations
> The original equations as specified by the user (i.e. before inserting point-currents into the membrane equation, before adding all the internally used variables and constants, etc.).

static spatialneuron_attribute(*neuron*, *name*)
> Selects a subtree from *SpatialNeuron* neuron and returns a *SpatialSubgroup*. If it does not exist, returns the *Group* attribute.

static spatialneuron_segment(*neuron*, *item*)
> Selects a segment from *SpatialNeuron* neuron, where item is a slice of either compartment indexes or distances. Note a: segment is not a *SpatialNeuron*, only a *Group*.

Tutorials and examples using this

- Example *compartmental/bipolar_cell*
- Example *compartmental/bipolar_with_inputs*
- Example *compartmental/bipolar_with_inputs2*
- Example *compartmental/cylinder*
- Example *compartmental/hh_with_spikes*
- Example *compartmental/hodgkin_huxley_1952*
- Example *compartmental/infinite_cable*
- Example *compartmental/lfp*
- Example *compartmental/morphotest*
- Example *compartmental/rall*
- Example *compartmental/spike_initiation*
- Example *frompapers/Brette_2012/Fig1*
- Example *frompapers/Brette_2012/Fig3AB*
- Example *frompapers/Brette_2012/Fig3CF*
- Example *frompapers/Brette_2012/Fig4*

- Example *frompapers/Brette_2012/Fig5A*

- Example *frompapers/Destexhe_et_al_1998*

SpatialStateUpdater(*args, **kw)	The *CodeRunner* that updates the state variables of a *SpatialNeuron* at every timestep.

SpatialStateUpdater class

(*Shortest import*: `from brian2.spatialneuron.spatialneuron import SpatialStateUpdater`)

class `brian2.spatialneuron.spatialneuron.`**SpatialStateUpdater**(*args*, **kw*)
    Bases: *brian2.groups.group.CodeRunner*, *brian2.groups.group.Group*

    The *CodeRunner* that updates the state variables of a *SpatialNeuron* at every timestep.

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.

Details

before_run(*run_namespace*)
    Optional method to prepare the object before a run.

    Called by *Network.after_run* before the main simulation loop starts.

SpatialSubgroup(*args, **kw)	A subgroup of a *SpatialNeuron*.

SpatialSubgroup class

(*Shortest import*: `from brian2.spatialneuron.spatialneuron import SpatialSubgroup`)

class `brian2.spatialneuron.spatialneuron.`**SpatialSubgroup**(*args*, **kw*)
    Bases: *brian2.groups.subgroup.Subgroup*

    A subgroup of a *SpatialNeuron*.

        Parameters source : int

            First compartment.

        stop : int

            Ending compartment, not included (as in slices).

        morphology : *Morphology*

            Morphology corresponding to the subgroup (not the full morphology).

        name : str, optional

            Name of the subgroup.

6.7.13 stateupdaters package

Module for transforming model equations into "abstract code" that can be then be further translated into executable code by the `codegen` module.

Exported members: *StateUpdateMethod*, *linear*, *exact*, *independent*, *milstein*, *heun*, *euler*, *rk2*, *rk4*, *ExplicitStateUpdater*, *exponential_euler*, *gsl_rk2*, *gsl_rk4*, *gsl_rkf45*, *gsl_rkck*, *gsl_rk8pd*

`GSL` module

Module containg the StateUpdateMethod for integration using the ODE solver provided in the GNU Scientific Library (GSL)

Exported members: *gsl_rk2*, *gsl_rk4*, *gsl_rkf45*, *gsl_rkck*, *gsl_rk8pd*

Classes

GSLContainer(method_options, integrator[, ...])	Class that contains information (equation- or integrator-related) required for later code generation

GSLContainer class

(*Shortest import*: `from brian2.stateupdaters.GSL import GSLContainer`)

class brian2.stateupdaters.GSL.**GSLContainer**(*method_options*, *integrator*, *abstract_code=None*, *needed_variables=[]*, *variable_flags=[]*)

> Bases: `object`
>
> Class that contains information (equation- or integrator-related) required for later code generation

Methods

__call__(obj)	Transfer the code object class saved in self to the object sent as an argument.
get_codeobj_class()	Return codeobject class based on target language and device.

Details

__call__(*obj*)

> Transfer the code object class saved in self to the object sent as an argument.
>
> This method is returned when calling *GSLStateUpdater*. This class inherits from *StateUpdateMethod* which orignally only returns abstract code. However, with GSL this returns a method because more is needed than just the abstract code: the state updater requires its own CodeObject that is different from the other *NeuronGroup* objects. This method adds this `CodeObject` to the `StateUpdater` object (and also adds the variables 't', 'dt', and other variables that are needed in the `GSLCodeGenerator`.
>
> > **Parameters obj** : *GSLStateUpdater*
> >
> > > the object that the codeobj_class and other variables need to be transferred to

Returns abstract_code : str

The abstract code (translated equations), that is returned conventionally by brian and used for later code generation in the `CodeGenerator.translate` method.

get_codeobj_class()

Return codeobject class based on target language and device.

Choose which version of the GSL `CodeObject` to use. If `isinstance(device, CPPStandaloneDevice)`, then we want the `GSLCPPStandaloneCodeObject`. Otherwise the return value is based on prefs.codegen.target.

Returns code_object : class

The respective `CodeObject` class (i.e. either `GSLCythonCodeObject` or `GSLCPPStandaloneCodeObject`).

GSLStateUpdater(integrator)	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

GSLStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.GSL import GSLStateUpdater`)

class brian2.stateupdaters.GSL.**GSLStateUpdater**(*integrator*)

Bases: *brian2.stateupdaters.base.StateUpdateMethod*

A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

New in version 2.1.

Methods

__call__(equations[, variables, method_options])	Translate equations to abstract_code.

Details

__call__(*equations*, *variables=None*, *method_options=None*)

Translate equations to abstract_code.

Parameters equations : *Equations*

object containing the equations that describe the ODE systemTransferClass(self)

variables : dict

dictionary containing str, `Variable` pairs

Returns method : callable

Method that needs to be called with `StateUpdater` to add CodeObject class and some other variables so these can be sent to the `CodeGenerator`

Objects

gsl_rk2	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

gsl_rk2 object

(*Shortest import*: `from brian2 import gsl_rk2`)

`brian2.stateupdaters.GSL.`**`gsl_rk2 = <brian2.stateupdaters.GSL.GSLStateUpdater object>`**

> A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.
>
> New in version 2.1.

gsl_rk4	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

gsl_rk4 object

(*Shortest import*: `from brian2 import gsl_rk4`)

`brian2.stateupdaters.GSL.`**`gsl_rk4 = <brian2.stateupdaters.GSL.GSLStateUpdater object>`**

> A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.
>
> New in version 2.1.

gsl_rk8pd	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

gsl_rk8pd object

(*Shortest import*: `from brian2 import gsl_rk8pd`)

`brian2.stateupdaters.GSL.`**`gsl_rk8pd = <brian2.stateupdaters.GSL.GSLStateUpdater object>`**

> A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.
>
> New in version 2.1.

gsl_rkck	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

gsl_rkck object

(*Shortest import*: `from brian2 import gsl_rkck`)

brian2.stateupdaters.GSL.**gsl_rkck = <brian2.stateupdaters.GSL.GSLStateUpdater object>**

> A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.
>
> New in version 2.1.

gsl_rkf45	A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.

gsl_rkf45 object

(*Shortest import*: `from brian2 import gsl_rkf45`)

brian2.stateupdaters.GSL.**gsl_rkf45 = <brian2.stateupdaters.GSL.GSLStateUpdater object>**

> A statupdater that rewrites the differential equations so that the GSL generator knows how to write the code in the target language.
>
> New in version 2.1.

base module

This module defines the *StateUpdateMethod* class that acts as a base class for all stateupdaters and allows to register stateupdaters so that it is able to return a suitable stateupdater object for a given set of equations. This is used for example in *NeuronGroup* when no state updater is given explicitly.

Exported members: *StateUpdateMethod*

Classes

StateUpdateMethod()	

Methods

StateUpdateMethod class

(*Shortest import*: `from brian2 import StateUpdateMethod`)

class brian2.stateupdaters.base.**StateUpdateMethod**

> Bases: *object*

Methods

___call___(equations[, variables, method_options])	Generate abstract code from equations.
_apply_stateupdater_(equations, variables, method)	Applies a given state updater to equations.
register(name, stateupdater)	Register a state updater.

Details

abstract __call__(*equations*, *variables=None*, *method_options=None*)

Generate abstract code from equations. The method also gets the the variables because some state updaters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing euler(eqs), for example.

> **Parameters equations** : *Equations*
>
> > The model equations.
>
> **variables** : dict, optional
>
> > The Variable objects for the model variables.
>
> **method_options** : dict, optional
>
> > Additional options specific to the state updater.
>
> **Returns** :
>
> **——-** :
>
> **code** : str
>
> > The abstract code performing a state update step.

static apply_stateupdater(*equations*, *variables*, *method*, *method_options=None*, *group_name=None*)

Applies a given state updater to equations. If a method is given, the state updater with the given name is used or if is a callable, then it is used directly. If a method is a list of names, all the methods will be tried until one that doesn't raise an *UnsupportedEquationsException* is found.

> **Parameters equations** : *Equations*
>
> > The model equations.
>
> **variables** : *dict*
>
> > The dictionary of Variable objects, describing the internal model variables.
>
> **method** : {callable, str, list of str}
>
> > A callable usable as a state updater, the name of a registered state updater or a list of names of state updaters.
>
> **Returns abstract_code** : str
>
> > The code integrating the given equations.

static register(*name*, *stateupdater*)

Register a state updater. Registered state updaters can be referred to via their name.

> **Parameters name** : str

A short name for the state updater (e.g. `'euler'`)

stateupdater : `StateUpdaterMethod`

The state updater object, e.g. an *`ExplicitStateUpdater`*.

`UnsupportedEquationsException`

UnsupportedEquationsException class

(*Shortest import*: `from brian2.stateupdaters.base import UnsupportedEquationsException`)

class `brian2.stateupdaters.base.`**`UnsupportedEquationsException`**

Bases: `Exception`

Functions

`extract_method_options`(method_options, ...)	Helper function to check `method_options` against options understood by this state updater, and setting default values for all unspecified options.

extract_method_options function

(*Shortest import*: `from brian2.stateupdaters.base import extract_method_options`)

`brian2.stateupdaters.base.`**`extract_method_options`**(*method_options*, *default_options*)

Helper function to check `method_options` against options understood by this state updater, and setting default values for all unspecified options.

Parameters **method_options** : dict or None

The options that the user specified for the state update.

default_options : dict

The default option values for this state updater (each admissible option needs to be present in this dictionary). To specify that a state updater does not take any options, provide an empty dictionary as the argument.

Returns **options** : dict

The final dictionary with all the options either at their default or at the user-specified value.

Raises

KeyError If the user specifies an option that is not understood by this state updater.

Examples

```
>>> options = extract_method_options({'a': True}, default_options={'b': False, 'c
↪': False})
Traceback (most recent call last):
...
KeyError: 'method_options specifies "a", but this is not an option for this state␣
↪updater. Avalaible options are: "b", "c".'
>>> options = extract_method_options({'a': True}, default_options={})
Traceback (most recent call last):
...
KeyError: 'method_options specifies "a", but this is not an option for this state␣
↪updater. This state updater does not accept any options.'
>>> options = extract_method_options({'a': True}, default_options={'a': False, 'b
↪': False})
>>> sorted(options.items())
[('a', True), ('b', False)]
```

exact module

Exact integration for linear equations.

Exported members: *linear*, *exact*, *independent*

Classes

IndependentStateUpdater()	A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations.

IndependentStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exact import IndependentStateUpdater`)

class `brian2.stateupdaters.exact.`**`IndependentStateUpdater`**
    Bases: *brian2.stateupdaters.base.StateUpdateMethod*

    A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations. The individual equations are solved by sympy.

    Deprecated since version 2.1: This method might be removed from future versions of Brian.

Methods

`__call__`(equations[, variables, method_options])	Generate abstract code from equations.

Details

__call__ (*equations*, *variables=None*, *method_options=None*)

Generate abstract code from equations. The method also gets the the variables because some state up-daters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing `euler(eqs)`, for example.

> **Parameters equations** : *Equations*
>
>> The model equations.
>
> **variables** : dict, optional
>
>> The `Variable` objects for the model variables.
>
> **method_options** : dict, optional
>
>> Additional options specific to the state updater.
>
> **Returns** :
>
> **——-** :
>
> **code** : str
>
>> The abstract code performing a state update step.

LinearStateUpdater()	A state updater for linear equations.

LinearStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exact import LinearStateUpdater`)

class brian2.stateupdaters.exact.**LinearStateUpdater**

Bases: *brian2.stateupdaters.base.StateUpdateMethod*

A state updater for linear equations. Derives a state updater step from the analytical solution given by sympy. Uses the matrix exponential (which is only implemented for diagonalizable matrices in sympy).

Methods

__call__(equations[, variables, method_options])	Generate abstract code from equations.

Details

__call__ (*equations*, *variables=None*, *method_options=None*)

Generate abstract code from equations. The method also gets the the variables because some state up-daters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing `euler(eqs)`, for example.

> **Parameters equations** : *Equations*

The model equations.

variables : dict, optional

The `Variable` objects for the model variables.

method_options : dict, optional

Additional options specific to the state updater.

Returns :

——- :

code : str

The abstract code performing a state update step.

Functions

[`get_linear_system`](eqs, variables)	Convert equations into a linear system using sympy.

get_linear_system function

(*Shortest import*: `from brian2.stateupdaters.exact import get_linear_system`)

`brian2.stateupdaters.exact.`**`get_linear_system`**(*eqs*, *variables*)
    Convert equations into a linear system using sympy.

> **Parameters eqs** : [*Equations*](...)
>
>> The model equations.
>
> **Returns (diff_eq_names, coefficients, constants)** : (list of str, `sympy.Matrix`, `sympy.Matrix`)
>
>> A tuple containing the variable names (`diff_eq_names`) corresponding to the rows of the matrix `coefficients` and the vector `constants`, representing the system of equations in the form M * X + B

Raises

`ValueError` If the equations cannot be converted into an M * X + B form.

Objects

[*exact*](...)	A state updater for linear equations.

exact object

(*Shortest import*: `from brian2 import exact`)

`brian2.stateupdaters.exact.`**`exact = LinearStateUpdater()`**
> A state updater for linear equations. Derives a state updater step from the analytical solution given by sympy. Uses the matrix exponential (which is only implemented for diagonalizable matrices in sympy).

independent	A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations.

independent object

(*Shortest import*: `from brian2 import independent`)

`brian2.stateupdaters.exact.`**`independent =`**
`<brian2.stateupdaters.exact.IndependentStateUpdater object>`
> A state update for equations that do not depend on other state variables, i.e. 1-dimensional differential equations. The individual equations are solved by sympy.
>
> Deprecated since version 2.1: This method might be removed from future versions of Brian.

linear	A state updater for linear equations.

linear object

(*Shortest import*: `from brian2 import linear`)

`brian2.stateupdaters.exact.`**`linear = LinearStateUpdater()`**
> A state updater for linear equations. Derives a state updater step from the analytical solution given by sympy. Uses the matrix exponential (which is only implemented for diagonalizable matrices in sympy).

explicit module

Numerical integration functions.

Exported members: *milstein*, *heun*, *euler*, *rk2*, *rk4*, *ExplicitStateUpdater*

Classes

ExplicitStateUpdater(description[, ...])	An object that can be used for defining state updaters via a simple description (see below).

ExplicitStateUpdater class

(*Shortest import*: `from brian2 import ExplicitStateUpdater`)

class `brian2.stateupdaters.explicit.`**`ExplicitStateUpdater`**(*description*, *stochastic=None*, *custom_check=None*)

Bases: *`brian2.stateupdaters.base.StateUpdateMethod`*

An object that can be used for defining state updaters via a simple description (see below). Resulting instances can be passed to the `method` argument of the *`NeuronGroup`* constructor. As other state updater functions the *`ExplicitStateUpdater`* objects are callable, returning abstract code when called with an *`Equations`* object.

A description of an explicit state updater consists of a (multi-line) string, containing assignments to variables and a final "x_new = …", stating the integration result for a single timestep. The assignments can be used to define an arbitrary number of intermediate results and can refer to `f(x, t)` (the function being integrated, as a function of `x`, the previous value of the state variable and `t`, the time) and `dt`, the size of the timestep.

For example, to define a Runge-Kutta 4 integrator (already provided as *`rk4`*), use:

```
k1 = dt*f(x,t)
k2 = dt*f(x+k1/2,t+dt/2)
k3 = dt*f(x+k2/2,t+dt/2)
k4 = dt*f(x+k3,t+dt)
x_new = x+(k1+2*k2+2*k3+k4)/6
```

Note that for stochastic equations, the function `f` only corresponds to the non-stochastic part of the equation. The additional function `g` corresponds to the stochastic part that has to be multiplied with the stochastic variable xi (a standard normal random variable – if the algorithm needs a random variable with a different variance/mean you have to multiply/add it accordingly). Equations with more than one stochastic variable do not have to be treated differently, the part referring to `g` is repeated for all stochastic variables automatically.

Stochastic integrators can also make reference to `dW` (a normal distributed random number with variance `dt`) and `g(x, t)`, the stochastic part of an equation. A stochastic state updater could therefore use a description like:

```
x_new = x + dt*f(x,t) + g(x, t) * dW
```

For simplicity, the same syntax is used for state updaters that only support additive noise, even though `g(x, t)` does not depend on `x` or `t` in that case.

There a some restrictions on the complexity of the expressions (but most can be worked around by using intermediate results as in the above Runge- Kutta example): Every statement can only contain the functions `f` and `g` once; The expressions have to be linear in the functions, e.g. you can use `dt*f(x, t)` but not `f(x, t)**2`.

> **Parameters description** : str
>
> > A state updater description (see above).
>
> **stochastic** : {None, 'additive', 'multiplicative'}
>
> > What kind of stochastic equations this state updater supports: `None` means no support of stochastic equations, `'additive'` means only equations with additive noise and `'multiplicative'` means supporting arbitrary stochastic equations.

Raises

ValueError If the parsing of the description failed.

See also:

euler, *rk2*, *rk4*, *milstein*

Notes

Since clocks are updated *after* the state update, the time `t` used in the state update step is still at its previous value. Enumerating the states and discrete times, `x_new = x + dt*f(x, t)` is therefore understood as $x_{i+1} = x_i + dt f(x_i, t_i)$, yielding the correct forward Euler integration. If the integrator has to refer to the time at the end of the timestep, simply use `t + dt` instead of `t`.

Attributes

DESCRIPTION	A complete state updater description
EXPRESSION	A single expression
OUTPUT	The last line of a state updater description
STATEMENT	An assignment statement
TEMP_VAR	Legal names for temporary variables

Methods

DESCRIPTION	A complete state updater description
EXPRESSION	A single expression
OUTPUT	The last line of a state updater description
STATEMENT	An assignment statement
TEMP_VAR	Legal names for temporary variables
__call__(eqs[, variables, method_options])	Apply a state updater description to model equations.
replace_func(x, t, expr, temp_vars, eq_symbols)	Used to replace a single occurance of `f(x, t)` or `g(x, t)`: `expr` is the non-stochastic (in the case of `f`) or stochastic part (`g`) of the expression defining the right-hand-side of the differential equation describing `var()`.

Details

**DESCRIPTION = {[Group:({~{'x_new'} W:(A-Z_a-z, 0-9A-Z_a-z) Suppress:('=')
rest of line})]... Group:({Suppress:('x_new') Suppress:('=') rest of
line})}**
    A complete state updater description

EXPRESSION = rest of line
    A single expression

OUTPUT = Group:({Suppress:('x_new') Suppress:('=') rest of line})
    The last line of a state updater description

STATEMENT = Group:({~{'x_new'} W:(A-Z_a-z, 0-9A-Z_a-z) Suppress:('=') rest of line})

An assignment statement

TEMP_VAR = {~{'x_new'} W:(A-Z_a-z, 0-9A-Z_a-z)}

Legal names for temporary variables

DESCRIPTION()

Requires all given `ParseExpression`s to be found in the given order. Expressions may be separated by whitespace. May be constructed using the `'+'` operator. May also be constructed using the `'-'` operator, which will suppress backtracking.

Example:

```
integer = Word(nums)
name_expr = OneOrMore(Word(alphas))

expr = And([integer("id"), name_expr("name"), integer("age")])
# more easily written as:
expr = integer("id") + name_expr("name") + integer("age")
```

EXPRESSION()

Token for matching strings that match a given regular expression. Defined with string specifying the regular expression in a form recognized by the stdlib Python re module. If the given regex contains named groups (defined using `(?P<name>...)`), these will be preserved as named `ParseResults`.

If instead of the Python stdlib `re` module you wish to use a different RE module (such as the `regex` module), you can do so by building your `Regex` object with a compiled RE that was compiled using `regex`.

Example:

```
realnum = Regex(r"[+-]?\d+\.\d*")
# ref: https://stackoverflow.com/questions/267399/how-do-you-match-only-valid-
↪roman-numerals-with-a-regular-expression
roman = Regex(r"M{0,4}(CM|CD|D?{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})")

# named fields in a regex will be returned as named results
date = Regex(r'(?P<year>\d{4})-(?P<month>\d\d?)-(?P<day>\d\d?)')

# the Regex class will accept re's compiled using the regex module
import regex
parser = pp.Regex(regex.compile(r'[0-9]'))
```

OUTPUT()

Converter to return the matched tokens as a list - useful for returning tokens of `ZeroOrMore` and `OneOrMore` expressions.

The optional `aslist` argument when set to True will return the parsed tokens as a Python list instead of a pyparsing ParseResults.

Example:

```
ident = Word(alphas)
num = Word(nums)
term = ident | num
func = ident + Opt(delimited_list(term))
print(func.parse_string("fn a, b, 100"))
# -> ['fn', 'a', 'b', '100']
```

(continues on next page)

```
func = ident + Group(Opt(delimited_list(term)))
print(func.parse_string("fn a, b, 100"))
# -> ['fn', ['a', 'b', '100']]
```

STATEMENT()

Converter to return the matched tokens as a list - useful for returning tokens of `ZeroOrMore` and `OneOr-More` expressions.

The optional `aslist` argument when set to True will return the parsed tokens as a Python list instead of a pyparsing ParseResults.

Example:

```
ident = Word(alphas)
num = Word(nums)
term = ident | num
func = ident + Opt(delimited_list(term))
print(func.parse_string("fn a, b, 100"))
# -> ['fn', 'a', 'b', '100']

func = ident + Group(Opt(delimited_list(term)))
print(func.parse_string("fn a, b, 100"))
# -> ['fn', ['a', 'b', '100']]
```

TEMP_VAR()

Requires all given `ParseExpression`s to be found in the given order. Expressions may be separated by whitespace. May be constructed using the `'+'` operator. May also be constructed using the `'-'` operator, which will suppress backtracking.

Example:

```
integer = Word(nums)
name_expr = OneOrMore(Word(alphas))

expr = And([integer("id"), name_expr("name"), integer("age")])
# more easily written as:
expr = integer("id") + name_expr("name") + integer("age")
```

__call__(*eqs*, *variables=None*, *method_options=None*)

Apply a state updater description to model equations.

> **Parameters** **eqs** : *Equations*
>
>> The equations describing the model
>
> **variables: dict-like, optional** :
>
>> The `Variable` objects for the model. Ignored by the explicit state updater.
>
> **method_options** : dict, optional
>
>> Additional options to the state updater (not used at the moment for the explicit state updaters).

Examples

```
>>> from brian2 import *
>>> eqs = Equations('dv/dt = -v / tau : volt')
>>> print(euler(eqs))
_v = -dt*v/tau + v
v = _v
>>> print(rk4(eqs))
__k_1_v = -dt*v/tau
__k_2_v = -dt*(__k_1_v/2 + v)/tau
__k_3_v = -dt*(__k_2_v/2 + v)/tau
__k_4_v = -dt*(__k_3_v + v)/tau
_v = __k_1_v/6 + __k_2_v/3 + __k_3_v/3 + __k_4_v/6 + v
v = _v
```

replace_func(*x*, *t*, *expr*, *temp_vars*, *eq_symbols*, *stochastic_variable=None*)

Used to replace a single occurance of `f(x, t)` or `g(x, t)`: `expr` is the non-stochastic (in the case of `f`) or stochastic part (`g`) of the expression defining the right-hand-side of the differential equation describing `var()`. It replaces the variable `var()` with the value given as `x` and `t` by the value given for `t`. Intermediate variables will be replaced with the appropriate replacements as well.

For example, in the *rk2* integrator, the second step involves the calculation of `f(k/2 + x, dt/2 + t)`. If `var()` is `v` and `expr` is `-v / tau`, this will result in `-(_k_v/2 + v)/tau`.

Note that this deals with only one state variable `var()`, given as an argument to the surrounding `_generate_RHS` function.

Functions

diagonal_noise(equations, variables)	Checks whether we deal with diagonal noise, i.e. one independent noise variable per variable.

diagonal_noise function

(*Shortest import*: `from brian2.stateupdaters.explicit import diagonal_noise`)

`brian2.stateupdaters.explicit.`**diagonal_noise**(*equations*, *variables*)

Checks whether we deal with diagonal noise, i.e. one independent noise variable per variable.

Raises

UnsupportedEquationsException If the noise is not diagonal.

split_expression(expr)	Split an expression into a part containing the function `f` and another one containing the function `g`.

split_expression function

(*Shortest import*: `from brian2.stateupdaters.explicit import split_expression`)

`brian2.stateupdaters.explicit.`**`split_expression`**(*expr*)

Split an expression into a part containing the function `f` and another one containing the function `g`. Returns a tuple of the two expressions (as sympy expressions).

> **Parameters expr** : str
>
> > An expression containing references to functions `f` and `g`.
>
> **Returns (non_stochastic, stochastic)** : tuple of sympy expressions
>
> > A pair of expressions representing the non-stochastic (containing function-independent terms and terms involving `f`) and the stochastic part of the expression (terms involving `g` and/or `dW`).

Examples

```
>>> split_expression('dt * __f(__x, __t)')
(dt*__f(__x, __t), None)
>>> split_expression('dt * __f(__x, __t) + __dW * __g(__x, __t)')
(dt*__f(__x, __t), __dW*__g(__x, __t))
>>> split_expression('1/(2*sqrt(dt))*(__g_support - __g(__x, __t))*(sqrt(__dW))')
(0, sqrt(__dW)*__g_support/(2*sqrt(dt)) - sqrt(__dW)*__g(__x, __t)/(2*sqrt(dt)))
```

Objects

euler	Forward Euler state updater

euler object

(*Shortest import*: `from brian2 import euler`)

`brian2.stateupdaters.explicit.`**`euler = ExplicitStateUpdater('''x_new = __dW*__g(__x, __t) + __x + dt*__f(__x, __t)''', stochastic='additive')`**

Forward Euler state updater

heun	Stochastic Heun method (for multiplicative Stratonovic SDEs with non-diagonal diffusion matrix)

heun object

(*Shortest import*: `from brian2 import heun`)

`brian2.stateupdaters.explicit.`**`heun = ExplicitStateUpdater('''__x_support = __dW*__g(__x, __t) + __x __g_support = __g(__x_support, __t + dt) x_new = 0.5*__dW*(__g_support + __g(__x, __t)) + __x + dt*__f(__x, __t)''', stochastic='multiplicative')`**

Stochastic Heun method (for multiplicative Stratonovic SDEs with non-diagonal diffusion matrix)

milstein	Derivative-free Milstein method

milstein object

(*Shortest import*: `from brian2 import milstein`)

`brian2.stateupdaters.explicit.`**`milstein = ExplicitStateUpdater('''__x_support = __x + dt**0.5*__g(__x, __t) + dt*__f(__x, __t) __g_support = __g(__x_support, __t) __k = __dW**2*(__g_support - __g(__x, __t))/(2*dt**0.5) x_new = __dW*__g(__x, __t) + __k + __x + dt*__f(__x, __t)''', stochastic='multiplicative')`**
> Derivative-free Milstein method

rk2	Second order Runge-Kutta method (midpoint method)

rk2 object

(*Shortest import*: `from brian2 import rk2`)

`brian2.stateupdaters.explicit.`**`rk2 = ExplicitStateUpdater('''__k = dt*__f(__x, __t) x_new = __x + dt*__f(__k/2 + __x, __t + dt/2)''', stochastic=None)`**
> Second order Runge-Kutta method (midpoint method)

rk4	Classical Runge-Kutta method (RK4)

rk4 object

(*Shortest import*: `from brian2 import rk4`)

`brian2.stateupdaters.explicit.`**`rk4 = ExplicitStateUpdater('''__k_1 = dt*__f(__x, __t) __k_2 = dt*__f(__k_1/2 + __x, __t + dt/2) __k_3 = dt*__f(__k_2/2 + __x, __t + dt/2) __k_4 = dt*__f(__k_3 + __x, __t + dt) x_new = __k_1/6 + __k_2/3 + __k_3/3 + __k_4/6 + __x''', stochastic=None)`**
> Classical Runge-Kutta method (RK4)

exponential_euler module

Exported members: *exponential_euler*

Classes

ExponentialEulerStateUpdater()	A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables).

ExponentialEulerStateUpdater class

(*Shortest import*: `from brian2.stateupdaters.exponential_euler import ExponentialEuler-StateUpdater`)

class `brian2.stateupdaters.exponential_euler.`**`ExponentialEulerStateUpdater`**
    Bases: *`brian2.stateupdaters.base.StateUpdateMethod`*

    A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables). Typical Hodgkin-Huxley equations fall into this category, it is therefore the default integration method used in the GENESIS simulator, for example.

Methods

`__call__`(equations[, variables, method_options])	Generate abstract code from equations.

Details

`__call__` (*equations*, *variables=None*, *method_options=None*)
    Generate abstract code from equations. The method also gets the the variables because some state updaters have to check whether variable names reflect other state variables (which can change from timestep to timestep) or are external values (which stay constant during a run) For convenience, this arguments are optional – this allows to directly see what code a state updater generates for a set of equations by simply writing `euler(eqs)`, for example.

> **Parameters equations** : *`Equations`*
>
>> The model equations.
>
> **variables** : dict, optional
>
>> The `Variable` objects for the model variables.
>
> **method_options** : dict, optional
>
>> Additional options specific to the state updater.
>
> **Returns** :
>
> **——-** :
>
> **code** : str
>
>> The abstract code performing a state update step.

Functions

`get_conditionally_linear_system`(eqs[, variables])	Convert equations into a linear system using sympy.

get_conditionally_linear_system function

(*Shortest import*: from brian2.stateupdaters.exponential_euler import get_conditionally_linear_system)

brian2.stateupdaters.exponential_euler.**get_conditionally_linear_system**(*eqs*, *variables=None*)

> Convert equations into a linear system using sympy.
>
> > **Parameters eqs** : *Equations*
> >
> > > The model equations.
> >
> > **Returns coefficients** : dict of (sympy expression, sympy expression) tuples
> >
> > > For every variable x, a tuple (M, B) containing the coefficients M and B (as sympy expressions) for M * x + B

Raises

ValueError If one of the equations cannot be converted into a M * x + B form.

Examples

```
>>> from brian2 import Equations
>>> eqs = Equations('''
... dv/dt = (-v + w**2.0) / tau : 1
... dw/dt = -w / tau : 1
... ''')
>>> system = get_conditionally_linear_system(eqs)
>>> print(system['v'])
(-1/tau, w**2.0/tau)
>>> print(system['w'])
(-1/tau, 0)
```

Objects

exponential_euler	A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables).

exponential_euler object

(*Shortest import*: from brian2 import exponential_euler)

brian2.stateupdaters.exponential_euler.**exponential_euler =**
<brian2.stateupdaters.exponential_euler.ExponentialEulerStateUpdater object>

> A state updater for conditionally linear equations, i.e. equations where each variable only depends linearly on itself (but possibly non-linearly on other variables). Typical Hodgkin-Huxley equations fall into this category, it is therefore the default integration method used in the GENESIS simulator, for example.

6.7.14 synapses package

Package providing synapse support.

Exported members: *Synapses*

parse_synaptic_generator_syntax module

Exported members: *parse_synapse_generator*

Functions

handle_range(*args, **kwds)	Checks the arguments/keywords for the range iterator

handle_range function

(*Shortest import*: `from brian2.synapses.parse_synaptic_generator_syntax import handle_range`)

`brian2.synapses.parse_synaptic_generator_syntax.`**`handle_range`**(*args*, **kwds*)

Checks the arguments/keywords for the range iterator

Should have 1-3 positional arguments.

Returns a dict with keys low, high, step. Default values are low=0, step=1.

handle_sample(*args, **kwds)	Checks the arguments/keywords for the sample iterator

handle_sample function

(*Shortest import*: `from brian2.synapses.parse_synaptic_generator_syntax import handle_sample`)

`brian2.synapses.parse_synaptic_generator_syntax.`**`handle_sample`**(*args*, **kwds*)

Checks the arguments/keywords for the sample iterator

Should have 1-3 positional arguments and 1 keyword argument (either p or size).

Returns a dict with keys `low, high, step, sample_size, p, size`. Default values are `low=0, step=1`. Sample size will be either ``'random' or 'fixed'`. In the first case, p will have a value and size will be `None` (and vice versa for the second case).

parse_synapse_generator(expr)	Returns a parsed form of a synapse generator expression.

parse_synapse_generator function

(*Shortest import*: from brian2.synapses.parse_synaptic_generator_syntax import parse_synapse_generator)

brian2.synapses.parse_synaptic_generator_syntax.**parse_synapse_generator**(*expr*)

Returns a parsed form of a synapse generator expression.

The general form is:

```
element for inner_variable in iterator_func(...)
```

or

```
element for inner_variable in iterator_func(...) if if_expression
```

Returns a dictionary with keys:

original_expression The original expression as a string.

element As above, a string expression.

inner_variable A variable name, as above.

iterator_func String. Either range or sample.

if_expression String expression or None.

iterator_kwds Dictionary of key/value pairs representing the keywords. See *handle_range* and *handle_sample*.

spikequeue module

The spike queue class stores future synaptic events produced by a given presynaptic neuron group (or postsynaptic for backward propagation in STDP).

Exported members: *SpikeQueue*

Classes

SpikeQueue(source_start, source_end)	Data structure saving the spikes and taking care of delays.

SpikeQueue class

(*Shortest import*: from brian2.synapses.spikequeue import SpikeQueue)

class brian2.synapses.spikequeue.**SpikeQueue**(*source_start*, *source_end*)

Bases: *object*

Data structure saving the spikes and taking care of delays.

> **Parameters** **source_start** : int
>
>> The start of the source indices (for subgroups)
>
> **source_end** : int
>
>> The end of the source indices (for subgroups)
>
> **Notes** :
>
> —— :

Data structure :

A spike queue is implemented as a 2D array `X` that is circular in the time :

direction (rows) and dynamic in the events direction (columns). The :

row index corresponding to the current timestep is `currentime`. :

Each element contains the target synapse index. :

Offsets :

Offsets are used to solve the problem of inserting multiple synaptic events :

with the same delay. This is difficult to vectorise. If there are n synaptic :

events with the same delay, these events are given an offset between 0 and :

n-1, corresponding to their relative position in the data structure. :

Attributes

_dt	The dt used for storing the spikes (will be set in *pre-pare*)
_source_end	The end of the source indices (for subgroups)
_source_start	The start of the source indices (for subgroups)
currenttime	The current time (in time steps)
n	number of events in each time step

Methods

advance()	Advances by one timestep
peek()	Returns the all the synaptic events corresponding to the current time, as an array of synapse indexes.
prepare(delays, dt, synapse_sources)	Prepare the data structures
push(sources)	Push spikes to the queue.

Details

_dt
> The dt used for storing the spikes (will be set in *prepare*)

_source_end
> The end of the source indices (for subgroups)

_source_start
> The start of the source indices (for subgroups)

currenttime
> The current time (in time steps)

n
> number of events in each time step

advance()
> Advances by one timestep

peek()
> Returns the all the synaptic events corresponding to the current time, as an array of synapse indexes.

prepare(*delays*, *dt*, *synapse_sources*)
> Prepare the data structures

> This is called every time the network is run. The size of the of the data structure (number of rows) is adjusted to fit the maximum delay in `delays`, if necessary. A flag is set if delays are homogeneous, in which case insertion will use a faster method implemented in `insert_homogeneous`.

push(*sources*)
> Push spikes to the queue.

> > **Parameters** **sources** : ndarray of int

> > > The indices of the neurons that spiked.

synapses module

Module providing the *Synapses* class and related helper classes/functions.

Exported members: *Synapses*

Classes

StateUpdater(*args, **kw)	The *CodeRunner* that updates the state variables of a *Synapses* at every timestep.

StateUpdater class

(*Shortest import*: `from brian2.synapses.synapses import StateUpdater`)

class brian2.synapses.synapses.**StateUpdater**(**args*, ***kw*)
> Bases: *brian2.groups.group.CodeRunner*

> The *CodeRunner* that updates the state variables of a *Synapses* at every timestep.

Methods

update_abstract_code(run_namespace)	Update the abstract code for the code object.

Details

update_abstract_code(*run_namespace*)
> Update the abstract code for the code object. Will be called in `before_run` and should update the `CodeRunner.abstract_code` attribute.

> Does nothing by default.

SummedVariableUpdater(*args, **kw)	The *CodeRunner* that updates a value in the target group with the sum over values in the *Synapses* object.

SummedVariableUpdater class

(*Shortest import*: `from brian2.synapses.synapses import SummedVariableUpdater`)

class brian2.synapses.synapses.**SummedVariableUpdater**(**args*, ***kw*)
    Bases: *[brian2.groups.group.CodeRunner](#)*

    The *[CodeRunner](#)* that updates a value in the target group with the sum over values in the *[Synapses](#)* object.

Methods

[before_run](#)(run_namespace)	Optional method to prepare the object before a run.

Details

before_run(*run_namespace*)
    Optional method to prepare the object before a run.

    Called by *[Network.after_run](#)* before the main simulation loop starts.

[Synapses](#)(*args, **kw)	Class representing synaptic connections.

Synapses class

(*Shortest import*: `from brian2 import Synapses`)

class brian2.synapses.synapses.**Synapses**(**args*, ***kw*)
    Bases: *[brian2.groups.group.Group](#)*

    Class representing synaptic connections.

    Creating a new *[Synapses](#)* object does by default not create any synapses, you have to call the *[Synapses.connect](#)* method for that.

    Parameters   source : *[SpikeSource](#)*

            The source of spikes, e.g. a *[NeuronGroup](#)*.

        target : *[Group](#)*, optional

            The target of the spikes, typically a *[NeuronGroup](#)*. If none is given, the same as source()

        model : `str`, *[Equations](#)*, optional

            The model equations for the synapses.

        on_pre : str, dict, optional

            The code that will be executed after every pre-synaptic spike. Can be either a single (possibly multi-line) string, or a dictionary mapping pathway names to code strings. In the first case, the pathway will be called `pre` and made available as an attribute of the same name. In the latter case, the given names will be used as the pathway/attribute names. Each pathway has its own code and its own delays.

        pre : str, dict, optional

Deprecated. Use `on_pre` instead.

on_post : str, dict, optional

The code that will be executed after every post-synaptic spike. Same conventions as for `on_pre`` `, the default name for the pathway is `post`.

post : str, dict, optional

Deprecated. Use `on_post` instead.

delay : *Quantity*, dict, optional

The delay for the "pre" pathway (same for all synapses) or a dictionary mapping pathway names to delays. If a delay is specified in this way for a pathway, it is stored as a single scalar value. It can still be changed afterwards, but only to a single scalar value. If you want to have delays that vary across synapses, do not use the keyword argument, but instead set the delays via the attribute of the pathway, e.g. `S.pre.delay = ...` (or `S.delay = ...` as an abbreviation), `S.post.delay = ...`, etc.

on_event : str or dict, optional

Define the events which trigger the pre and post pathways. By default, both pathways are triggered by the `'spike'` event, i.e. the event that is triggered by the `threshold` condition in the connected groups.

multisynaptic_index : str, optional

The name of a variable (which will be automatically created) that stores the "synapse number". This number enumerates all synapses between the same source and target so that they can be distinguished. For models where each source-target pair has only a single connection, this number only wastes memory (it would always default to 0), it is therefore not stored by default. Defaults to `None` (no variable).

namespace : dict, optional

A dictionary mapping identifier names to objects. If not given, the namespace will be filled in at the time of the call of *Network.run*, with either the values from the `namespace` argument of the *Network.run* method or from the local context, if no such argument is given.

dtype : `dtype`, dict, optional

The `numpy.dtype` that will be used to store the values, or a dictionary specifying the type for variable names. If a value is not provided for a variable (or no value is provided at all), the preference setting *core.default_float_dtype* is used.

codeobj_class : class, optional

The `CodeObject` class to use to run code.

dt : *Quantity*, optional

The time step to be used for the update of the state variables. Cannot be combined with the `clock` argument.

clock : *Clock*, optional

The update clock to be used. If neither a clock, nor the `dt` argument is specified, the *defaultclock* will be used.

order : int, optional

The priority of of this group for operations occurring at the same time step and in the same scheduling slot. Defaults to 0.

> **method** : str, *StateUpdateMethod*, optional
>
> > The numerical integration method to use. If none is given, an appropriate one is automatically determined.
>
> **name** : str, optional
>
> > The name for this object. If none is given, a unique name of the form `synapses`, `synapses_1`, etc. will be automatically chosen.

Attributes

N_incoming_post	The number of incoming synapses for each neuron in the post-synaptic group.
N_outgoing_pre	The number of outgoing synapses for each neuron in the pre-synaptic group.
_connect_called	remember whether connect was called to raise an error if an assignment to a synaptic variable is attempted without a preceding connect.
_pathways	List of all *SynapticPathway* objects
_registered_variables	Set of `Variable` objects that should be resized when the number of synapses changes
_synaptic_updaters	List of names of all updaters, e.g.
delay	The presynaptic delay (if a pre-synaptic pathway exists).
*delay_*	The presynaptic delay without unit information (if apre-synaptic pathway exists).
events	"Events" for all the pathways
state_updater	Performs numerical integration step
subexpression_updater	Update the "constant over a time step" subexpressions
summed_updaters	"Summed variable" mechanism -- sum over all synapses of a pre-/postsynaptic target

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.
check_variable_write(variable)	Checks that *Synapses.connect* has been called before setting a synaptic variable.
connect([condition, i, j, p, n, ...])	Add synapses.
register_variable(variable)	Register a `DynamicArray` to be automatically resized when the size of the indices change.
unregister_variable(variable)	Unregister a `DynamicArray` from the automatic resizing mechanism.
verify_dependencies(eq, eq_type, deps, ...)	Helper function to verify that event-driven equations do not depend on clock-driven equations and the other way round.

Details

N_incoming_post
The number of incoming synapses for each neuron in the post-synaptic group.

N_outgoing_pre
The number of outgoing synapses for each neuron in the pre-synaptic group.

_connect_called
remember whether connect was called to raise an error if an assignment to a synaptic variable is attempted without a preceding connect.

_pathways
List of all *SynapticPathway* objects

_registered_variables
Set of Variable objects that should be resized when the number of synapses changes

_synaptic_updaters
List of names of all updaters, e.g. ['pre', 'post']

delay
The presynaptic delay (if a pre-synaptic pathway exists).

delay_
The presynaptic delay without unit information (if apre-synaptic pathway exists).

events
"Events" for all the pathways

state_updater
Performs numerical integration step

subexpression_updater
Update the "constant over a time step" subexpressions

summed_updaters
"Summed variable" mechanism – sum over all synapses of a pre-/postsynaptic target

before_run(*run_namespace*)
Optional method to prepare the object before a run.

Called by *Network.after_run* before the main simulation loop starts.

check_variable_write(*variable*)
Checks that *Synapses.connect* has been called before setting a synaptic variable.

> **Parameters variable** : Variable
>
>> The variable that the user attempts to set.

Raises

> **TypeError** If *Synapses.connect* has not been called yet.

connect(*condition=None*, *i=None*, *j=None*, *p=1.0*, *n=1*, *skip_if_invalid=False*, *namespace=None*, *level=0*)
Add synapses.

See *Synapses* for details.

> **Parameters condition** : str, bool, optional

A boolean or string expression that evaluates to a boolean. The expression can depend on indices `i` and `j` and on pre- and post-synaptic variables. Can be combined with arguments `n`, and `p` but not `i` or `j`.

i : int, ndarray of int, str, optional

The presynaptic neuron indices It can be an index or array of indices if combined with the `j` argument, or it can be a string generator expression.

j : int, ndarray of int, str, optional

The postsynaptic neuron indices. It can be an index or array of indices if combined with the `i` argument, or it can be a string generator expression.

p : float, str, optional

The probability to create `n` synapses wherever the `condition` evaluates to true. Cannot be used with generator syntax for `j`.

n : int, str, optional

The number of synapses to create per pre/post connection pair. Defaults to 1.

skip_if_invalid : bool, optional

If set to True, rather than raising an error if you try to create an invalid/out of range pair (i, j) it will just quietly skip those synapses.

namespace : dict-like, optional

A namespace that will be used in addition to the group-specific namespaces (if defined). If not specified, the locals and globals around the run function will be used.

level : int, optional

How deep to go up the stack frame to look for the locals/global (see `namespace` argument).

Examples

```
>>> from brian2 import *
>>> import numpy as np
>>> G = NeuronGroup(10, 'dv/dt = -v / tau : 1', threshold='v>1', reset='v=0')
>>> S = Synapses(G, G, 'w:1', on_pre='v+=w')
>>> S.connect(condition='i != j') # all-to-all but no self-connections
>>> S.connect(i=0, j=0) # connect neuron 0 to itself
>>> S.connect(i=np.array([1, 2]), j=np.array([2, 1])) # connect 1->2 and 2->1
>>> S.connect() # connect all-to-all
>>> S.connect(condition='i != j', p=0.1)  # Connect neurons with 10%␣
↪probability, exclude self-connections
>>> S.connect(j='i', n=2)   # Connect all neurons to themselves with 2 synapses
>>> S.connect(j='k for k in range(i+1)') # Connect neuron i to all j with 0<=j␣
↪<=i
>>> S.connect(j='i+(-1)**k for k in range(2) if i>0 and i<N_pre-1') # connect␣
↪neuron i to its neighbours if it has both neighbours
>>> S.connect(j='k for k in sample(N_post, p=i*1.0/(N_pre-1))') # neuron i␣
↪connects to j with probability i/(N-1)
>>> S.connect(j='k for k in sample(N_post, size=i//2)') # Each neuron␣
↪connects to i//2 other neurons (chosen randomly)
```

register_variable(*variable*)

> Register a `DynamicArray` to be automatically resized when the size of the indices change. Called automatically when a `SynapticArrayVariable` specifier is created.

unregister_variable(*variable*)

> Unregister a `DynamicArray` from the automatic resizing mechanism.

static verify_dependencies(*eq*, *eq_type*, *deps*, *should_not_depend_on*,
                                          should_not_depend_on_name)

> Helper function to verify that event-driven equations do not depend on clock-driven equations and the other way round.

> > **Parameters** **eq** : `SingleEquation`
> >
> > > The equation to verify
> >
> > **eq_type** : str
> >
> > > The type of the equation (for the error message)
> >
> > **deps** : list
> >
> > > A list of dependencies
> >
> > **should_not_depend_on** : list
> >
> > > A list of equations to verify against the dependencies
> >
> > **should_not_depend_on_name** : str
> >
> > > The name of the list of equations (for the error message)

Raises

:exc:`EquationError` If the given equation depends on something in the other set of equations.

Tutorials and examples using this

- Tutorial *1-intro-to-brian-neurons*
- Tutorial *2-intro-to-brian-synapses*
- Tutorial *3-intro-to-brian-simulations*
- Example *COBAHH*
- Example *CUBA*
- Example *adaptive_threshold*
- Example *advanced/custom_events*
- Example *advanced/float_32_64_benchmark*
- Example *compartmental/bipolar_with_inputs*
- Example *compartmental/bipolar_with_inputs2*
- Example *compartmental/lfp*
- Example *frompapers/Brette_2012/Fig5A*
- Example *frompapers/Brunel_Hakim_1999*
- Example *frompapers/Brunel_Wang_2001*

- Example *frompapers/Clopath_et_al_2010_homeostasis*
- Example *frompapers/Clopath_et_al_2010_no_homeostasis*
- Example *frompapers/Diesmann_et_al_1999*
- Example *frompapers/Izhikevich_2007*
- Example *frompapers/Kremer_et_al_2011_barrel_cortex*
- Example *frompapers/Stimberg_et_al_2018/example_1_COBA*
- Example *frompapers/Stimberg_et_al_2018/example_2_gchi_astrocyte*
- Example *frompapers/Stimberg_et_al_2018/example_3_io_synapse*
- Example *frompapers/Stimberg_et_al_2018/example_4_rsmean*
- Example *frompapers/Stimberg_et_al_2018/example_4_synrel*
- Example *frompapers/Stimberg_et_al_2018/example_5_astro_ring*
- Example *frompapers/Stimberg_et_al_2018/example_6_COBA_with_astro*
- Example *frompapers/Sturzl_et_al_2000*
- Example *frompapers/Vogels_et_al_2011*
- Example *standalone/STDP_standalone*
- Example *standalone/cuba_openmp*
- Example *standalone/standalone_multiplerun*
- Example *synapses/STDP*
- Example *synapses/continuous_interaction*
- Example *synapses/efficient_gaussian_connectivity*
- Example *synapses/gapjunctions*
- Example *synapses/jeffress*
- Example *synapses/licklider*
- Example *synapses/nonlinear*
- Example *synapses/spatial_connections*
- Example *synapses/spike_based_homeostasis*
- Example *synapses/state_variables*
- Example *synapses/synapses*

`SynapticIndexing`(synapses)

Methods

SynapticIndexing class

(*Shortest import*: `from brian2.synapses.synapses import SynapticIndexing`)

class `brian2.synapses.synapses.`**`SynapticIndexing`**(*synapses*)
  Bases: `object`

Methods

`__call__`([index, index_var])	Returns synaptic indices for `index`, which can be a tuple of indices (including arrays and slices), a single index or a string.

Details

`__call__`(*index=None*, *index_var='_idx'*)
  Returns synaptic indices for `index`, which can be a tuple of indices (including arrays and slices), a single index or a string.

`SynapticPathway`(*args, **kw)	The `CodeRunner` that applies the pre/post statement(s) to the state variables of synapses where the pre-/postsynaptic group spiked in this time step.

SynapticPathway class

(*Shortest import*: `from brian2.synapses.synapses import SynapticPathway`)

class `brian2.synapses.synapses.`**`SynapticPathway`**(*args*, **kw*)
  Bases: *brian2.groups.group.CodeRunner*, *brian2.groups.group.Group*

The `CodeRunner` that applies the pre/post statement(s) to the state variables of synapses where the pre-/postsynaptic group spiked in this time step.

Parameters synapses : *Synapses*

  Reference to the main *Synapses* object

prepost : {'pre', 'post'}

  Whether this object should react to pre- or postsynaptic spikes

objname : str, optional

  The name to use for the object, will be appendend to the name of `synapses` to create a name in the sense of *Nameable*. If `None` is provided (the default), `prepost` will be used.

delay : *Quantity*, optional

  A scalar delay (same delay for all synapses) for this pathway. If not given, delays are expected to vary between synapses.

Attributes

_initialise_queue_codeobj	The `CodeObject` initalising the `SpikeQueue` at the begin of a run
queue	The `SpikeQueue`

Methods

before_run(run_namespace)	Optional method to prepare the object before a run.
check_variable_write(variable)	Function that can be overwritten to raise an error if writing to a variable should not be allowed.
create_code_objects(run_namespace)	
initialise_queue()	
push_spikes()	
update_abstract_code([run_namespace, level])	Update the abstract code for the code object.

Details

_initialise_queue_codeobj
   The `CodeObject` initalising the `SpikeQueue` at the begin of a run

queue
   The `SpikeQueue`

before_run(*run_namespace*)
   Optional method to prepare the object before a run.

   Called by *Network.after_run* before the main simulation loop starts.

check_variable_write(*variable*)
   Function that can be overwritten to raise an error if writing to a variable should not be allowed. Note that this does *not* deal with incorrect writes that are general to all kind of variables (incorrect units, writing to a read-only variable, etc.). This function is only used for type-specific rules, e.g. for raising an error in *Synapses* when writing to a synaptic variable before any *connect* call.

   By default this function does nothing.

   Parameters   **variable** : `Variable`

   The variable that the user attempts to set.

create_code_objects(*run_namespace*)

initialise_queue()

push_spikes()

update_abstract_code(*run_namespace=None*, *level=0*)
   Update the abstract code for the code object. Will be called in *before_run* and should update the `CodeRunner.abstract_code` attribute.

   Does nothing by default.

SynapticSubgroup(synapses, indices)	A simple subgroup of *Synapses* that can be used for indexing.

SynapticSubgroup class

(*Shortest import*: `from brian2.synapses.synapses import SynapticSubgroup`)

class brian2.synapses.synapses.**SynapticSubgroup**(*synapses*, *indices*)
    Bases: object

A simple subgroup of *Synapses* that can be used for indexing.

> **Parameters** **indices** : ndarray of int
>
> > The synaptic indices represented by this subgroup.
>
> **synaptic_pre** : DynamicArrayVariable
>
> > References to all pre-synaptic indices. Only used to throw an error when new synapses where added after creating this object.

Functions

find_synapses(index, synaptic_neuron)	

find_synapses function

(*Shortest import*: `from brian2.synapses.synapses import find_synapses`)

brian2.synapses.synapses.**find_synapses**(*index*, *synaptic_neuron*)

slice_to_test(x)	Returns a testing function corresponding to whether an index is in slice x.

slice_to_test function

(*Shortest import*: `from brian2.synapses.synapses import slice_to_test`)

brian2.synapses.synapses.**slice_to_test**(*x*)
    Returns a testing function corresponding to whether an index is in slice x. x can also be an int.

6.7.15 units package

The unit system.

Exported members: pamp, namp, uamp, mamp, amp, kamp, Mamp, Gamp, Tamp, kelvin, kilogram, pmetre, nmetre, umetre, mmetre, metre, kmetre, Mmetre, Gmetre, Tmetre, pmeter, nmeter, umeter, mmeter, meter … (218 more members)

`allunits` **module**

THIS FILE IS AUTOMATICALLY GENERATED BY A STATIC CODE GENERATION TOOL DO NOT EDIT BY HAND

Instead edit the template:

>   dev/tools/static_codegen/units_template.py

Exported members: `metre`, `meter`, `kilogram`, `second`, `amp`, `ampere`, `kelvin`, `mole`, `mol`, `candle`, `kilo-gramme`, `gram`, `gramme`, `molar`, `radian`, `steradian`, `hertz`, `newton`, `pascal`, `joule`, `watt`, `coulomb`, `volt`, `farad`, `ohm` … (2045 more members)

Objects

celsius	A dummy object to raise errors when `celsius` is used.

celsius object

(*Shortest import*: `from brian2.units.allunits import celsius`)

brian2.units.allunits.**celsius = <brian2.units.allunits._Celsius object>**
>   A dummy object to raise errors when `celsius` is used. The use of *celsius* can lead to ambiguities when mixed with temperatures in `kelvin`, so its use is no longer supported. See github issue #817 for details.

`constants` **module**

A module providing some physical units as *Quantity* objects. Note that these units are not imported by wildcard imports (e.g. `from brian2 import *`), they have to be imported explicitly. You can use `import ... as ...` to import them with shorter names, e.g.:

```python
from brian2.units.constants import faraday_constant as F
```

The available constants are:

Constant	Symbol(s)	Brian name	Value
Avogadro constant	N_A, L	`avogadro_constant`	$6.022140857 \times 10^{23} \, \text{mol}^{-1}$
Boltzmann constant	k	`boltzmann_constant`	$1.38064852 \times 10^{-23} \, \text{J K}^{-1}$
Electric constant	ϵ_0	`electric_constant`	$8.854187817 \times 10^{-12} \, \text{F m}^{-1}$
Electron mass	m_e	`electron_mass`	$9.10938356 \times 10^{-31} \, \text{kg}$
Elementary charge	e	`elementary_charge`	$1.6021766208 \times 10^{-19} \, \text{C}$
Faraday constant	F	`faraday_constant`	$96485.33289 \, \text{C mol}^{-1}$
Gas constant	R	`gas_constant`	$8.3144598 \, \text{J mol}^{-1} \text{K}^{-1}$
Magnetic constant	μ_0	`magnetic_constant`	$12.566370614 \times 10^{-7} \, \text{N A}^{-2}$
Molar mass constant	M_u	`molar_mass_constant`	$1 \times 10^{-3} \, \text{kg mol}^{-1}$
0°C		`zero_celsius`	$273.15 \, \text{K}$

`fundamentalunits` module

Defines physical units and quantities

Quantity	Unit	Symbol
Length	metre	m
Mass	kilogram	kg
Time	second	s
Electric current	ampere	A
Temperature	kelvin	K
Quantity of substance	mole	mol
Luminosity	candle	cd

Exported members: `DimensionMismatchError`, `get_or_create_dimension()`, `get_dimensions()`, `is_dimensionless()`, `have_same_dimensions()`, `in_unit()`, `in_best_unit()`, `Quantity`, `Unit`, `register_new_unit()`, `check_units()`, `is_scalar_type()`, `get_unit()`

Classes

`Dimension`(dims)	Stores the indices of the 7 basic SI unit dimension (length, mass, etc.).

Dimension class

(*Shortest import*: `from brian2.units.fundamentalunits import Dimension`)

class `brian2.units.fundamentalunits.`**`Dimension`**(*dims*)
> Bases: `object`

> Stores the indices of the 7 basic SI unit dimension (length, mass, etc.).

> Provides a subset of arithmetic operations appropriate to dimensions: multiplication, division and powers, and equality testing.

> > **Parameters dims** : sequence of `float`

> > > The dimension indices of the 7 basic SI unit dimensions.

Notes

Users shouldn't use this class directly, it is used internally in Quantity and Unit. Even internally, never use `Dimension(...)` to create a new instance, use `get_or_create_dimension()` instead. This function makes sure that only one Dimension instance exists for every combination of indices, allowing for a very fast dimensionality check with `is`.

Attributes

dim	Returns the *Dimension* object itself.
is_dimensionless	Whether this Dimension is dimensionless.

Methods

get_dimension(d)	Return a specific dimension.

Details

dim

> Returns the *Dimension* object itself. This can be useful, because it allows to check for the dimension of an object by checking its dim attribute – this will return a *Dimension* object for *Quantity*, *Unit* and *Dimension*.

is_dimensionless

> Whether this Dimension is dimensionless.

Notes

> Normally, instead one should check dimension for being identical to *DIMENSIONLESS*.

get_dimension(*d*)

> Return a specific dimension.
>
> > **Parameters d** : str
> >
> > > A string identifying the SI basic unit dimension. Can be either a description like "length" or a basic unit like "m" or "metre".
> >
> > **Returns dim** : float
> >
> > > The dimensionality of the dimension d.

DimensionMismatchError(description, *dims)	Exception class for attempted operations with inconsistent dimensions.

DimensionMismatchError class

(*Shortest import*: from brian2 import DimensionMismatchError)

class brian2.units.fundamentalunits.**DimensionMismatchError**(*description*, **dims*)

> Bases: Exception

Exception class for attempted operations with inconsistent dimensions.

For example, 3*mvolt + 2*amp raises this exception. The purpose of this class is to help catch errors based on incorrect units. The exception will print a representation of the dimensions of the two inconsistent objects that were operated on.

> **Parameters description** : str

A description of the type of operation being performed, e.g. Addition, Multiplication, etc.

dims : *Dimension*

The physical dimensions of the objects involved in the operation, any number of them is possible

Tutorials and examples using this

- Tutorial *1-intro-to-brian-neurons*

Quantity(arr[, dim, dtype, copy, force_quantity])	A number with an associated physical dimension.

Quantity class

(*Shortest import*: `from brian2 import Quantity`)

class `brian2.units.fundamentalunits.`**Quantity** (*arr*, *dim=None*, *dtype=None*, *copy=False*, *force_quantity=False*)

Bases: `numpy.ndarray`, `object`

A number with an associated physical dimension. In most cases, it is not necessary to create a Quantity object by hand, instead use multiplication and division of numbers with the constant unit names `second`, `kilogram`, etc.

See also:

Unit

Notes

The *Quantity* class defines arithmetic operations which check for consistency of dimensions and raise the DimensionMismatchError exception if they are inconsistent. It also defines default and other representations for a number for printing purposes.

See the documentation on the Unit class for more details about the available unit names like mvolt, etc.

Casting rules

The rules that define the casting operations for Quantity object are:

1. Quantity op Quantity = Quantity Performs dimension checking if appropriate

2. (Scalar or Array) op Quantity = Quantity Assumes that the scalar or array is dimensionless

There is one exception to the above rule, the number `0` is interpreted as having "any dimension".

Examples

```
>>> from brian2 import *
>>> I = 3 * amp # I is a Quantity object
>>> R = 2 * ohm # same for R
>>> I * R
6. * volt
>>> (I * R).in_unit(mvolt)
'6000. mV'
>>> (I * R) / mvolt
6000.0
>>> X = I + R
Traceback (most recent call last):
    ...
DimensionMismatchError: Addition, dimensions were (A) (m^2 kg s^-3 A^-2)
>>> Is = np.array([1, 2, 3]) * amp
>>> Is * R
array([ 2.,  4.,  6.]) * volt
>>> np.asarray(Is * R) # gets rid of units
array([ 2.,  4.,  6.])
```

Attributes

dimensions	The physical dimensions of this quantity.
is_dimensionless	Whether this is a dimensionless quantity.
dim	The physical dimensions of this quantity.

Methods

with_dimensions(value, *args, **keywords)	Create a *Quantity* object with dim.
has_same_dimensions(other)	Return whether this object has the same dimensions as another.
in_unit(u[, precision, python_code])	Represent the quantity in a given unit.
in_best_unit([precision, python_code])	Represent the quantity in the "best" unit.

Details

dimensions
    The physical dimensions of this quantity.

is_dimensionless
    Whether this is a dimensionless quantity.

dim

static with_dimensions (*value*, **args*, ***keywords*)
    Create a *Quantity* object with dim.

> **Parameters value** : {array_like, number}
>
>> The value of the dimension
>
>> **args** : {*Dimension*, sequence of float}

Either a single argument (a *Dimension*) or a sequence of 7 values.

kwds :

Keywords defining the dim, see *Dimension* for details.

Returns q : *Quantity*

A *Quantity* object with the given dim

Examples

All of these define an equivalent *Quantity* object:

```
>>> from brian2 import *
>>> Quantity.with_dimensions(2, get_or_create_dimension(length=1))
2. * metre
>>> Quantity.with_dimensions(2, length=1)
2. * metre
>>> 2 * metre
2. * metre
```

has_same_dimensions(*other*)
Return whether this object has the same dimensions as another.

Parameters other : {*Quantity*, array-like, number}

The object to compare the dimensions against.

Returns same : `bool`

`True` if `other` has the same dimensions.

in_unit(*u*, *precision=None*, *python_code=False*)
Represent the quantity in a given unit. If `python_code` is `True`, this will return valid python code, i.e. a string like `5.0 * um ** 2` instead of `5.0 um^2`

Parameters u : {*Quantity*, *Unit*}

The unit in which to show the quantity.

precision : `int`, optional

The number of digits of precision (in the given unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

python_code : `bool`, optional

Whether to return valid python code (`True`) or a human readable string (`False`, the default).

Returns s : `str`

String representation of the object in unit `u`.

See also:

in_unit()

Examples

```
>>> from brian2.units import *
>>> from brian2.units.stdunits import *
>>> x = 25.123456 * mV
>>> x.in_unit(volt)
'0.02512346 V'
>>> x.in_unit(volt, 3)
'0.025 V'
>>> x.in_unit(mV, 3)
'25.123 mV'
```

in_best_unit (*precision=None*, *python_code=False*, **regs*)

Represent the quantity in the "best" unit.

Parameters **python_code** : `bool`, optional

If set to `False` (the default), will return a string like `5.0  um^2` which is not a valid Python expression. If set to `True`, it will return `5.0 * um ** 2` instead.

precision : `int`, optional

The number of digits of precision (in the best unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.

regs : `UnitRegistry` objects

The registries where to search for units. If none are given, the standard, user-defined and additional registries are searched in that order.

Returns **representation** : `str`

A string representation of this `Quantity`.

See also:

in_best_unit()

Examples

```
>>> from brian2.units import *
```

```
>>> x = 0.00123456 * volt
```

```
>>> x.in_best_unit()
'1.23456 mV'
```

```
>>> x.in_best_unit(3)
'1.235 mV'
```

Tutorials and examples using this

- Example *frompapers/Destexhe_et_al_1998*

- Example *frompapers/Platkiewicz_Brette_2011*

- Example *frompapers/Stimberg_et_al_2018/example_1_COBA*

Unit(arr[, dim, scale, name, dispname, ...])	A physical unit.

Unit class

(*Shortest import*: `from brian2 import Unit`)

class brian2.units.fundamentalunits.**Unit**(*arr*, *dim=None*, *scale=0*, *name=None*, *dispname=None*, *latexname=None*, *iscompound=False*, *dtype=None*, *copy=False*)

Bases: *brian2.units.fundamentalunits.Quantity*

A physical unit.

Normally, you do not need to worry about the implementation of units. They are derived from the *Quantity* object with some additional information (name and string representation).

Basically, a unit is just a number with given dimensions, e.g. mvolt = 0.001 with the dimensions of voltage. The units module defines a large number of standard units, and you can also define your own (see below).

The unit class also keeps track of various things that were used to define it so as to generate a nice string representation of it. See below.

When creating scaled units, you can use the following prefixes:

Factor	Name	Prefix
10^24	yotta	Y
10^21	zetta	Z
10^18	exa	E
10^15	peta	P
10^12	tera	T
10^9	giga	G
10^6	mega	M
10^3	kilo	k
10^2	hecto	h
10^1	deka	da
1		
10^-1	deci	d
10^-2	centi	c
10^-3	milli	m
10^-6	micro	u (mu in SI)
10^-9	nano	n
10^-12	pico	p
10^-15	femto	f
10^-18	atto	a
10^-21	zepto	z
10^-24	yocto	y

Defining your own

It can be useful to define your own units for printing purposes. So for example, to define the newton metre, you write

```
>>> from brian2 import *
>>> from brian2.units.allunits import newton
>>> Nm = newton * metre
```

You can then do

```
>>> (1*Nm).in_unit(Nm)
'1. N m'
```

New "compound units", i.e. units that are composed of other units will be automatically registered and from then on used for display. For example, imagine you define total conductance for a membrane, and the total area of that membrane:

```
>>> conductance = 10.*nS
>>> area = 20000*um**2
```

If you now ask for the conductance density, you will get an "ugly" display in basic SI dimensions, as Brian does not know of a corresponding unit:

```
>>> conductance/area
0.5 * metre ** -4 * kilogram ** -1 * second ** 3 * amp ** 2
```

By using an appropriate unit once, it will be registered and from then on used for display when appropriate:

```
>>> usiemens/cm**2
usiemens / (cmetre ** 2)
>>> conductance/area  # same as before, but now Brian knows about uS/cm^2
50. * usiemens / (cmetre ** 2)
```

Note that user-defined units cannot override the standard units (volt, second, etc.) that are predefined by Brian. For example, the unit Nm has the dimensions "length^2·mass/time^2", and therefore the same dimensions as the standard unit joule. The latter will be used for display purposes:

```
>>> 3*joule
3. * joule
>>> 3*Nm
3. * joule
```

Attributes

_dispname	The display name of this unit.
_latexname	A LaTeX expression for the name of this unit.
_name	The full name of this unit.
dim	The Dimensions of this unit
dispname	The display name of the unit
iscompound	Whether this unit is a combination of other units.
latexname	The LaTeX name of the unit
name	The name of the unit

continues on next page

Table 465 – continued from previous page	
scale	The scale for this unit (as the integer exponent of 10), i.e. a scale of 3 means 10^3, e.g.

Methods

create(dim, name, dispname[, latexname, scale])	Create a new named unit.
create_scaled_unit(baseunit, scalefactor)	Create a scaled unit from a base unit.
set_display_name(name)	Sets the display name for the unit.
set_latex_name(name)	Sets the LaTeX name for the unit.
set_name(name)	Sets the name for the unit.

Details

_dispname
> The display name of this unit.

_latexname
> A LaTeX expression for the name of this unit.

_name
> The full name of this unit.

dim
> The Dimensions of this unit

dispname
> The display name of the unit

iscompound
> Whether this unit is a combination of other units.

latexname
> The LaTeX name of the unit

name
> The name of the unit

scale
> The scale for this unit (as the integer exponent of 10), i.e. a scale of 3 means 10^3, e.g. for a "k" prefix.

static create(*dim*, *name*, *dispname*, *latexname=None*, *scale=0*)
> Create a new named unit.

> > **Parameters dim** : *Dimension*

> > > The dimensions of the unit.

> > **name** : str

> > > The full name of the unit, e.g. `'volt'`

> > **dispname** : str

> > > The display name, e.g. `'V'`

> > **latexname** : str, optional

The name as a LaTeX expression (math mode is assumed, do not add $ signs or similar), e.g. `'\omega'`. If no `latexname` is specified, `dispname` will be used.

scale : int, optional

The scale of this unit as an exponent of 10, e.g. -3 for a unit that is 1/1000 of the base scale. Defaults to 0 (i.e. a base unit).

Returns u : `Unit`

The new unit.

static create_scaled_unit(*baseunit*, *scalefactor*)

Create a scaled unit from a base unit.

Parameters baseunit : `Unit`

The unit of which to create a scaled version, e.g. `volt`, `amp`.

scalefactor : `str`

The scaling factor, e.g. `"m"` for mvolt, mamp

Returns u : `Unit`

The new unit.

set_display_name(*name*)

Sets the display name for the unit.

Deprecated since version 2.1: Create a new unit with `Unit.create` instead.

set_latex_name(*name*)

Sets the LaTeX name for the unit.

Deprecated since version 2.1: Create a new unit with `Unit.create` instead.

set_name(*name*)

Sets the name for the unit.

Deprecated since version 2.1: Create a new unit with `Unit.create` instead.

`UnitRegistry`()	Stores known units for printing in best units.

UnitRegistry class

(*Shortest import*: `from brian2.units.fundamentalunits import UnitRegistry`)

class `brian2.units.fundamentalunits.`**UnitRegistry**

Bases: `object`

Stores known units for printing in best units.

All a user needs to do is to use the `register_new_unit()` function.

Default registries:

The units module defines three registries, the standard units, user units, and additional units. Finding best units is done by first checking standard, then user, then additional. New user units are added by using the `register_new_unit()` function.

Standard units includes all the basic non-compound unit names built in to the module, including volt, amp, etc. Additional units defines some compound units like newton metre (Nm) etc.

Methods

add(u)	Add a unit to the registry
__getitem__(x)	Returns the best unit for quantity x

Details

add(u)

   Add a unit to the registry

__getitem__(x)

   Returns the best unit for quantity x

   The algorithm is to consider the value:

   m=abs(x/u)

   for all matching units u. We select the unit where this ratio is the closest to 10 (if it is an array with several values, we select the unit where the deviations from that are the smallest. More precisely, the unit that minimizes the sum of (log10(m)-1)**2 over all entries).

Functions

check_units(**au)	Decorator to check units of arguments passed to a function

check_units function

(*Shortest import*: `from brian2 import check_units`)

`brian2.units.fundamentalunits.`**`check_units`**(***au*)

   Decorator to check units of arguments passed to a function

Raises

DimensionMismatchError In case the input arguments or the return value do not have the expected dimensions.

TypeError If an input argument or return value was expected to be a boolean but is not.

Notes

This decorator will destroy the signature of the original function, and replace it with the signature (`*args`, `**kwds`). Other decorators will do the same thing, and this decorator critically needs to know the signature of the function it is acting on, so it is important that it is the first decorator to act on a function. It cannot be used in combination with another decorator that also needs to know the signature of the function.

Note that the `bool` type is "strict", i.e. it expects a proper boolean value and does not accept 0 or 1. This is not the case the other way round, declaring an argument or return value as "1" *does* allow for a `True` or `False` value.

Examples

```
>>> from brian2.units import *
>>> @check_units(I=amp, R=ohm, wibble=metre, result=volt)
... def getvoltage(I, R, **k):
...     return I*R
```

You don't have to check the units of every variable in the function, and you can define what the units should be for variables that aren't explicitly named in the definition of the function. For example, the code above checks that the variable wibble should be a length, so writing

```
>>> getvoltage(1*amp, 1*ohm, wibble=1)
Traceback (most recent call last):
...
DimensionMismatchError: Function "getvoltage" variable "wibble" has wrong␣
→dimensions, dimensions were (1) (m)
```

fails, but

```
>>> getvoltage(1*amp, 1*ohm, wibble=1*metre)
1. * volt
```

passes. String arguments or None are not checked

```
>>> getvoltage(1*amp, 1*ohm, wibble='hello')
1. * volt
```

By using the special name result, you can check the return value of the function.

You can also use 1 or bool as a special value to check for a unitless number or a boolean value, respectively:

```
>>> @check_units(value=1, absolute=bool, result=bool)
... def is_high(value, absolute=False):
...     if absolute:
...         return abs(value) >= 5
...     else:
...         return value >= 5
```

This will then again raise an error if the argument if not of the expected type:

```
>>> is_high(7)
True
>>> is_high(-7, True)
True
>>> is_high(3, 4)
Traceback (most recent call last):
...
TypeError: Function "is_high" expected a boolean value for argument "absolute"␣
→but got 4.
```

If the return unit depends on the unit of an argument, you can also pass a function that takes the units of all the arguments as its inputs (in the order specified in the function header):

```
>>> @check_units(result=lambda d: d**2)
... def square(value):
...     return value**2
```

If several arguments take arbitrary units but they have to be consistent among each other, you can state the name of another argument as a string to state that it uses the same unit as that argument.

```
>>> @check_units(summand_1=None, summand_2='summand_1')
... def multiply_sum(multiplicand, summand_1, summand_2):
...     "Calculates multiplicand*(summand_1 + summand_2)"
...     return multiplicand*(summand_1 + summand_2)
>>> multiply_sum(3, 4*mV, 5*mV)
27. * mvolt
>>> multiply_sum(3*nA, 4*mV, 5*mV)
27. * pwatt
>>> multiply_sum(3*nA, 4*mV, 5*nA)
Traceback (most recent call last):
...
brian2.units.fundamentalunits.DimensionMismatchError: Function 'multiply_sum'␣
→expected the same arguments for arguments 'summand_1', 'summand_2', but␣
→argument 'summand_1' has unit V, while argument 'summand_2' has unit A.
```

fail_for_dimension_mismatch(obj1[, obj2, ...])	Compare the dimensions of two objects.

fail_for_dimension_mismatch function

(*Shortest import*: from brian2.units.fundamentalunits import fail_for_dimension_mismatch)

brian2.units.fundamentalunits.**fail_for_dimension_mismatch**(*obj1*, *obj2=None*, *error_message=None*, ***error_quantities*)

Compare the dimensions of two objects.

> **Parameters** **obj1, obj2** : {array-like, *Quantity*}
>
> > The object to compare. If `obj2` is `None`, assume it to be dimensionless
>
> **error_message** : str, optional
>
> > An error message that is used in the DimensionMismatchError
>
> **error_quantities** : dict mapping str to *Quantity*, optional
>
> > Quantities in this dictionary will be converted using the `_short_str` helper method and inserted into the `error_message` (which should have placeholders with the corresponding names). The reason for doing this in a somewhat complicated way instead of directly including all the details in `error_messsage` is that converting large quantity arrays to strings can be rather costly and we don't want to do it if no error occured.
>
> **Returns** **dim1, dim2** : *Dimension*, *Dimension*
>
> > The physical dimensions of the two arguments (so that later code does not need to get the dimensions again).

Raises

DimensionMismatchError If the dimensions of `obj1` and `obj2` do not match (or, if `obj2` is `None`, in case `obj1` is not dimensionless).

Notes

Implements special checking for `0`, treating it as having "any dimensions".

get_dimensions(obj)	Return the dimensions of any object that has them.

get_dimensions function

(*Shortest import*: `from brian2 import get_dimensions`)

`brian2.units.fundamentalunits.`**`get_dimensions`**(*obj*)

Return the dimensions of any object that has them.

Slightly more general than *Quantity.dimensions* because it will return *DIMENSIONLESS* if the object is of number type but not a *Quantity* (e.g. a `float` or `int`).

> **Parameters obj** : `object`
>
>> The object to check.
>
> **Returns dim** : *Dimension*
>
>> The physical dimensions of the `obj`.

get_or_create_dimension(*args, **kwds)	Create a new Dimension object or get a reference to an existing one.

get_or_create_dimension function

(*Shortest import*: `from brian2 import get_or_create_dimension`)

`brian2.units.fundamentalunits.`**`get_or_create_dimension`**(**args*, ***kwds*)

Create a new Dimension object or get a reference to an existing one. This function takes care of only creating new objects if they were not created before and otherwise returning a reference to an existing object. This allows to compare dimensions very efficiently using `is`.

> **Parameters args** : sequence of `float`
>
>> A sequence with the indices of the 7 elements of an SI dimension.
>
> **kwds** : keyword arguments
>
>> a sequence of `keyword=value` pairs where the keywords are the names of the SI dimensions, or the standard unit.

Notes

The 7 units are (in order):

Length, Mass, Time, Electric Current, Temperature, Quantity of Substance, Luminosity

and can be referred to either by these names or their SI unit names, e.g. length, metre, and m all refer to the same thing here.

Examples

The following are all definitions of the dimensions of force

```
>>> from brian2 import *
>>> get_or_create_dimension(length=1, mass=1, time=-2)
metre * kilogram * second ** -2
>>> get_or_create_dimension(m=1, kg=1, s=-2)
metre * kilogram * second ** -2
>>> get_or_create_dimension([1, 1, -2, 0, 0, 0, 0])
metre * kilogram * second ** -2
```

get_unit(d)	Find an unscaled unit (e.g.

get_unit function

(*Shortest import*: `from brian2 import get_unit`)

`brian2.units.fundamentalunits.`**`get_unit`**(*d*)

Find an unscaled unit (e.g. `volt` but not `mvolt`) for a *Dimension*.

> **Parameters d** : *Dimension*
>
> > The dimension to find a unit for.
>
> **Returns u** : *Unit*
>
> > A registered unscaled *Unit* for the dimensions d, or a new *Unit* if no unit was found.

get_unit_for_display(d)	Return a string representation of an appropriate unscaled unit or `'1'` for a dimensionless quantity.

get_unit_for_display function

(*Shortest import*: `from brian2.units.fundamentalunits import get_unit_for_display`)

`brian2.units.fundamentalunits.`**`get_unit_for_display`**(*d*)

Return a string representation of an appropriate unscaled unit or `'1'` for a dimensionless quantity.

> **Parameters d** : *Dimension* or int
>
> > The dimension to find a unit for.
>
> **Returns s** : str
>
> > A string representation of the respective unit or the string `'1'`.

have_same_dimensions(obj1, obj2)	Test if two values have the same dimensions.

have_same_dimensions function

(*Shortest import*: `from brian2 import have_same_dimensions`)

`brian2.units.fundamentalunits.`**`have_same_dimensions`**(*obj1*, *obj2*)

Test if two values have the same dimensions.

> **Parameters obj1, obj2** : {*Quantity*, array-like, number}
>
>> The values of which to compare the dimensions.
>
> **Returns same** : `bool`
>
>> True if `obj1` and `obj2` have the same dimensions.

in_best_unit(x[, precision])	Represent the value in the "best" unit.

in_best_unit function

(*Shortest import*: `from brian2 import in_best_unit`)

`brian2.units.fundamentalunits.`**`in_best_unit`**(*x*, *precision=None*)

Represent the value in the "best" unit.

> **Parameters x** : {*Quantity*, array-like, number}
>
>> The value to display
>
> **precision** : `int`, optional
>
>> The number of digits of precision (in the best unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.
>
> **Returns representation** : `str`
>
>> A string representation of this *Quantity*.

See also:

Quantity.in_best_unit

Examples

```
>>> from brian2.units import *
>>> in_best_unit(0.00123456 * volt)
'1.23456 mV'
>>> in_best_unit(0.00123456 * volt, 2)
'1.23 mV'
>>> in_best_unit(0.123456)
'0.123456'
>>> in_best_unit(0.123456, 2)
'0.12'
```

in_unit(x, u[, precision])	Display a value in a certain unit with a given precision.

in_unit function

(*Shortest import*: `from brian2 import in_unit`)

brian2.units.fundamentalunits.**in_unit**(*x*, *u*, *precision=None*)

Display a value in a certain unit with a given precision.

> **Parameters** **x** : {*Quantity*, array-like, number}
>
> > The value to display
>
> **u** : {*Quantity*, *Unit*}
>
> > The unit to display the value `x` in.
>
> **precision** : `int`, optional
>
> > The number of digits of precision (in the given unit, see Examples). If no value is given, numpy's `get_printoptions()` value is used.
>
> **Returns** **s** : `str`
>
> > A string representation of `x` in units of `u`.

See also:

Quantity.in_unit

Examples

```
>>> from brian2 import *
>>> in_unit(3 * volt, mvolt)
'3000. mV'
>>> in_unit(123123 * msecond, second, 2)
'123.12 s'
>>> in_unit(10 * uA/cm**2, nA/um**2)
'1.00000000e-04 nA/(um^2)'
>>> in_unit(10 * mV, ohm * amp)
'0.01 ohm A'
>>> in_unit(10 * nS, ohm)
...
Traceback (most recent call last):
    ...
DimensionMismatchError: Non-matching unit for method "in_unit",
dimensions were (m^-2 kg^-1 s^3 A^2) (m^2 kg s^-3 A^-2)
```

is_dimensionless(obj)	Test if a value is dimensionless or not.

is_dimensionless function

(*Shortest import*: `from brian2 import is_dimensionless`)

brian2.units.fundamentalunits.**is_dimensionless**(*obj*)
   Test if a value is dimensionless or not.

> **Parameters obj** : `object`
>
> > The object to check.
>
> **Returns dimensionless** : `bool`
>
> > True if `obj` is dimensionless.

is_scalar_type(obj)	Tells you if the object is a 1d number type.

is_scalar_type function

(*Shortest import*: `from brian2 import is_scalar_type`)

brian2.units.fundamentalunits.**is_scalar_type**(*obj*)
   Tells you if the object is a 1d number type.

> **Parameters obj** : `object`
>
> > The object to check.
>
> **Returns scalar** : `bool`
>
> > True if `obj` is a scalar that can be interpreted as a dimensionless *Quantity*.

quantity_with_dimensions(floatval, dims)	Create a new *Quantity* with the given dimensions.

quantity_with_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import quantity_with_dimensions`)

brian2.units.fundamentalunits.**quantity_with_dimensions**(*floatval*, *dims*)
   Create a new *Quantity* with the given dimensions. Calls `get_or_create_dimensions` with the dimension tuple of the `dims` argument to make sure that unpickling (which calls this function) does not accidentally create new Dimension objects which should instead refer to existing ones.

> **Parameters floatval** : `float`
>
> > The floating point value of the quantity.
>
> **dims** : *Dimension*
>
> > The physical dimensions of the quantity.
>
> **Returns q** : *Quantity*
>
> > A quantity with the given dimensions.

See also:

`get_or_create_dimensions`

Examples

```
>>> from brian2 import *
>>> quantity_with_dimensions(0.001, volt.dim)
1. * mvolt
```

register_new_unit(u)	Register a new unit for automatic displaying of quantities

register_new_unit function

(*Shortest import*: `from brian2 import register_new_unit`)

`brian2.units.fundamentalunits.`**`register_new_unit`**(*u*)

Register a new unit for automatic displaying of quantities

> **Parameters** **u** : *Unit*
>
> > The unit that should be registered.

Examples

```
>>> from brian2 import *
>>> 2.0*farad/metre**2
2. * metre ** -4 * kilogram ** -1 * second ** 4 * amp ** 2
>>> register_new_unit(pfarad / mmetre**2)
>>> 2.0*farad/metre**2
2000000. * pfarad / (mmetre ** 2)
```

wrap_function_change_dimensions(func, ...)	Returns a new function that wraps the given function `func` so that it changes the dimensions of its input.

wrap_function_change_dimensions function

(*Shortest import*: `from brian2.units.fundamentalunits import wrap_function_change_dimensions`)

`brian2.units.fundamentalunits.`**`wrap_function_change_dimensions`**(*func,*
change_dim_func)

Returns a new function that wraps the given function `func` so that it changes the dimensions of its input. Quantities are transformed to unitless numpy arrays before calling `func`, the output is a quantity with the original dimensions passed through the function `change_dim_func`. A typical use would be a `sqrt` function that uses `lambda d: d ** 0.5` as `change_dim_func`.

These transformations apply only to the very first argument, all other arguments are ignored/untouched.

wrap_function_dimensionless(func)	Returns a new function that wraps the given function `func` so that it raises a DimensionMismatchError if the function is called on a quantity with dimensions (excluding dimensionless quantities).

wrap_function_dimensionless function

(*Shortest import*: from brian2.units.fundamentalunits import wrap_function_dimensionless)

brian2.units.fundamentalunits.**wrap_function_dimensionless**(*func*)

Returns a new function that wraps the given function func so that it raises a DimensionMismatchError if the function is called on a quantity with dimensions (excluding dimensionless quantities). Quantities are transformed to unitless numpy arrays before calling func.

These checks/transformations apply only to the very first argument, all other arguments are ignored/untouched.

wrap_function_keep_dimensions(func)	Returns a new function that wraps the given function func so that it keeps the dimensions of its input.

wrap_function_keep_dimensions function

(*Shortest import*: from brian2.units.fundamentalunits import wrap_function_keep_dimensions)

brian2.units.fundamentalunits.**wrap_function_keep_dimensions**(*func*)

Returns a new function that wraps the given function func so that it keeps the dimensions of its input. Quantities are transformed to unitless numpy arrays before calling func, the output is a quantity with the original dimensions re-attached.

These transformations apply only to the very first argument, all other arguments are ignored/untouched, allowing to work functions like sum to work as expected with additional axis etc. arguments.

wrap_function_remove_dimensions(func)	Returns a new function that wraps the given function func so that it removes any dimensions from its input.

wrap_function_remove_dimensions function

(*Shortest import*: from brian2.units.fundamentalunits import wrap_function_remove_dimensions)

brian2.units.fundamentalunits.**wrap_function_remove_dimensions**(*func*)

Returns a new function that wraps the given function func so that it removes any dimensions from its input. Useful for functions that are returning integers (indices) or booleans, irrespective of the datatype contained in the array.

These transformations apply only to the very first argument, all other arguments are ignored/untouched.

Objects

DIMENSIONLESS	The singleton object for dimensionless Dimensions.

DIMENSIONLESS object

(*Shortest import*: `from brian2.units.fundamentalunits import DIMENSIONLESS`)

brian2.units.fundamentalunits.**DIMENSIONLESS = Dimension()**
> The singleton object for dimensionless Dimensions.

additional_unit_register	*UnitRegistry* containing additional units (newton*metre, farad / metre, ...)

additional_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import additional_unit_register`)

brian2.units.fundamentalunits.**additional_unit_register =**
<brian2.units.fundamentalunits.UnitRegistry object>
> *UnitRegistry* containing additional units (newton*metre, farad / metre, …)

standard_unit_register	*UnitRegistry* containing all the standard units (metre, kilogram, um2...)

standard_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import standard_unit_register`)

brian2.units.fundamentalunits.**standard_unit_register =**
<brian2.units.fundamentalunits.UnitRegistry object>
> *UnitRegistry* containing all the standard units (metre, kilogram, um2…)

user_unit_register	*UnitRegistry* containing all units defined by the user

user_unit_register object

(*Shortest import*: `from brian2.units.fundamentalunits import user_unit_register`)

brian2.units.fundamentalunits.**user_unit_register =**
<brian2.units.fundamentalunits.UnitRegistry object>
> *UnitRegistry* containing all units defined by the user

stdunits module

Optional short unit names

This module defines the following short unit names:

mV, mA, uA (micro_amp), nA, pA, mF, uF, nF, nS, mS, uS, ms, Hz, kHz, MHz, cm, cm2, cm3, mm, mm2, mm3, um, um2, um3

Exported members: `mV`, `mA`, `uA`, `nA`, `pA`, `pF`, `uF`, `nF`, `nS`, `uS`, `mS`, `ms`, `us`, `Hz`, `kHz`, `MHz`, `cm`, `cm2`, `cm3`, `mm`, `mm2`, `mm3`, `um`, `um2`, `um3` … (3 more members)

unitsafefunctions **module**

Unit-aware replacements for numpy functions.

Exported members: `log()`, `log10()`, `exp()`, `expm1()`, `log1p()`, `exprel()`, `sin()`, `cos()`, `tan()`, `arcsin()`, `arccos()`, `arctan()`, `sinh()`, `cosh()`, `tanh()`, `arcsinh()`, `arccosh()`, `arctanh()`, `diagonal()`, `ravel()`, `trace()`, `dot()`, `where()`, `ones_like()`, `zeros_like()` … (2 more members)

Functions

`arange`([start,] stop[, step,][, dtype])	Return evenly spaced values within a given interval.

arange **function**

(*Shortest import*: `from brian2 import arange`)

`brian2.units.unitsafefunctions.`**`arange`**`([start ], stop[, step ], dtype=None)`

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including `start` but excluding `stop()`). For integer arguments the function is equivalent to the Python built-in `range` function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

> **Parameters** **start** : number, optional
>
> > Start of interval. The interval includes this value. The default start value is 0.
>
> **stop** : number
>
> > End of interval. The interval does not include this value, except in some cases where `step()` is not an integer and floating point round-off affects the length of `out`.
>
> **step** : number, optional
>
> > Spacing between values. For any output `out`, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If `step()` is specified as a position argument, `start` must also be given.
>
> **dtype** : dtype
>
> > The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.
>
> **Returns** **arange** : ndarray
>
> > Array of evenly spaced values.
> >
> > For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of `out` being greater than `stop()`.

> See also:

> `numpy.linspace` Evenly spaced numbers with careful handling of endpoints.

> `numpy.ogrid` Arrays of evenly spaced numbers in N-dimensions.

> `numpy.mgrid` Grid-shaped arrays of evenly spaced numbers in N-dimensions.

Examples

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

arccos(x, /[, out, where, casting, order, ...])	Trigonometric inverse cosine, element-wise.

arccos function

(*Shortest import*: `from brian2 import arccos`)

`brian2.units.unitsafefunctions.`**`arccos`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*]*)*

Trigonometric inverse cosine, element-wise.

The inverse of `cos()` so that, if `y = cos(x)`, then `x = arccos(y)`.

Parameters **x** : array_like

x-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns **angle** : ndarray

The angle of the ray intersecting the unit circle at the given x-coordinate in radians [0, pi]. This is a scalar if x is a scalar.

See also:

`cos()`, `arctan()`, `arcsin()`, `emath.arccos`

Notes

arccos() is a multivalued function: for each x there are infinitely many numbers z such that `cos(z) = x`. The convention is to return the angle z whose real part lies in `[0, pi]`.

For real-valued input data types, *arccos()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arccos()* is a complex analytic function that has branch cuts `[-inf, -1]` and `[1, inf]` and is continuous from above on the former and from below on the latter.

The inverse *cos()* is also known as acos or cos^-1.

References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```

arccosh(x, /[, out, where, casting, order, ...])	Inverse hyperbolic cosine, element-wise.

arccosh function

(*Shortest import*: `from brian2 import arccosh`)

`brian2.units.unitsafefunctions.`**`arccosh`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*])

Inverse hyperbolic cosine, element-wise.

> **Parameters x** : array_like
>
>> Input array.
>
>> **out** : ndarray, None, or tuple of ndarray and None, optional
>
>>> A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
>
>> **where** : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns arccosh : ndarray

Array of the same shape as `x`. This is a scalar if `x` is a scalar.

See also:

`cosh()`, `arcsinh()`, `sinh()`, `arctanh()`, `tanh()`

Notes

`arccosh()` is a multivalued function: for each `x` there are infinitely many numbers `z` such that `cosh(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi, pi]` and the real part in `[0, inf]`.

For real-valued input data types, `arccosh()` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, `arccosh()` is a complex analytical function that has a branch cut `[-inf, 1]` and is continuous from above on it.

References

[R13], [R14]

Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

| `arcsin`(x, /[, out, where, casting, order, ...]) | Inverse sine, element-wise. |

arcsin function

(*Shortest import*: `from brian2 import arcsin`)

`brian2.units.unitsafefunctions.`**`arcsin`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*])

Inverse sine, element-wise.

Parameters x : array_like

`y`-coordinate on the unit circle.

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns angle : ndarray

The inverse sine of each element in x, in radians and in the closed interval `[-pi/2, pi/2]`. This is a scalar if x is a scalar.

See also:

`sin()`, `cos()`, `arccos()`, `tan()`, `arctan()`, `arctan2`, `emath.arcsin`

Notes

`arcsin()` is a multivalued function: for each x there are infinitely many numbers z such that $sin(z) = x$. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, `arcsin()` is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as `asin` or sin^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

Examples

```
>>> np.arcsin(1)      # pi/2
1.5707963267948966
>>> np.arcsin(-1)     # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

| `arcsinh`(x, /[, out, where, casting, order, ...]) | Inverse hyperbolic sine element-wise. |

arcsinh function

(*Shortest import*: `from brian2 import arcsinh`)

brian2.units.unitsafefunctions.**arcsinh**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

> Inverse hyperbolic sine element-wise.
>
> > **Parameters** **x** : array_like
> >
> > > Input array.
> >
> > **out** : ndarray, None, or tuple of ndarray and None, optional
> >
> > > A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
> >
> > **where** : array_like, optional
> >
> > > This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
> >
> > ****kwargs** :
> >
> > > For other keyword-only arguments, see the ufunc docs.
> >
> > **Returns** **out** : ndarray or scalar
> >
> > > Array of the same shape as `x`. This is a scalar if `x` is a scalar.

Notes

`arcsinh()` is a multivalued function: for each x there are infinitely many numbers z such that `sinh(z) = x`. The convention is to return the z whose imaginary part lies in `[-pi/2, pi/2]`.

For real-valued input data types, `arcsinh()` always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the `invalid` floating point error flag.

For complex-valued input, `arccos()` is a complex analytical function that has branch cuts `[1j, infj]` and `[-1j, -infj]` and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as `asinh` or `sinh^-1`.

References

[R15], [R16]

Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

---

arctan(x, /[, out, where, casting, order, ...])	Trigonometric inverse tangent, element-wise.

---

arctan function

(*Shortest import*: `from brian2 import arctan`)

brian2.units.unitsafefunctions.**arctan**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*])

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

> **Parameters** **x** : array_like
>
> > **out** : ndarray, None, or tuple of ndarray and None, optional
> >
> > > A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
> >
> > **where** : array_like, optional
> >
> > > This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
> >
> > ****kwargs** :
> >
> > > For other keyword-only arguments, see the ufunc docs.
>
> **Returns** **out** : ndarray or scalar
>
> > Out has the same shape as `x`. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). This is a scalar if `x` is a scalar.

> See also:

> **arctan2** The "four quadrant" arctan of the angle formed by (x, y) and the positive x-axis.

> **angle()** Argument of complex values.

Notes

arctan() is a multi-valued function: for each x there are infinitely many numbers z such that tan(z) = x. The convention is to return the angle z whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arctan()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *arctan()* is a complex analytic function that has [1j, infj] and [-1j, -infj] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as `atan` or tan^{-1}.

References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

Examples

We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

```
>>> np.pi/4
0.78539816339744828
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```

arctanh(x, /[, out, where, casting, order, ...])	Inverse hyperbolic tangent element-wise.

arctanh function

(*Shortest import*: `from brian2 import arctanh`)

`brian2.units.unitsafefunctions.`**`arctanh`**(*x, /, out=None, *, where=True, casting='same_kind',*
                                    order='K', dtype=None, subok=True[, *signature, extobj*])

> Inverse hyperbolic tangent element-wise.
>
> > **Parameters** **x** : array_like
> >
> > > Input array.
> >
> > **out** : ndarray, None, or tuple of ndarray and None, optional
> >
> > > A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
> >
> > **where** : array_like, optional
> >
> > > This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
> >
> > ****kwargs** :
> >
> > > For other keyword-only arguments, see the ufunc docs.
> >
> > **Returns** **out** : ndarray or scalar
> >
> > > Array of the same shape as `x`. This is a scalar if `x` is a scalar.

See also:

`emath.arctanh`

Notes

`arctanh()` is a multivalued function: for each x there are infinitely many numbers z such that `tanh(z) = x`. The convention is to return the z whose imaginary part lies in `[-pi/2, pi/2]`.

For real-valued input data types, `arctanh()` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, `arctanh()` is a complex analytical function that has branch cuts `[-1, -inf]` and `[1, inf]` and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as `atanh` or `tanh^-1`.

References

[R17], [R18]

Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

`cos`(x, /[, out, where, casting, order, ...])	Cosine element-wise.

cos function

(*Shortest import*: `from brian2 import cos`)

`brian2.units.unitsafefunctions.`**`cos`**(*x*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

Cosine element-wise.

> **Parameters** **x** : array_like
>
>> Input array in radians.
>>
>> **out** : ndarray, None, or tuple of ndarray and None, optional
>>
>>> A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
>>
>> **where** : array_like, optional
>>
>>> This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
>>
>> ****kwargs** :
>>
>>> For other keyword-only arguments, see the ufunc docs.

Returns y : ndarray

The corresponding cosine values. This is a scalar if x is a scalar.

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

$cosh$(x, /[, out, where, casting, order, ...])	Hyperbolic cosine, element-wise.

cosh function

(*Shortest import*: from brian2 import cosh)

brian2.units.unitsafefunctions.**cosh**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*])

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (np.exp(x) + np.exp(-x))` and `np.cos(1j*x)`.

Parameters x : array_like

Input array.

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its origi-

nal value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns out : ndarray or scalar

Output array of same shape as `x`. This is a scalar if `x` is a scalar.

Examples

```
>>> np.cosh(0)
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
```

diagonal(a[, offset, axis1, axis2])	Return specified diagonals.

diagonal function

(*Shortest import*: `from brian2 import diagonal`)

brian2.units.unitsafefunctions.**diagonal**(*a*, *offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals.

If `a` is 2-D, returns the diagonal of `a` with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If `a` has more than two dimensions, then the axes specified by `axis1` and `axis2` are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing `axis1` and `axis2` and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning is issued.

Starting in NumPy 1.9 it returns a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In some future release, it will return a read/write view and writing to the returned array will alter your original array. The returned array will have the same type as the input array.

If you don't write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diagonal(a).copy()` instead of just `np.diagonal(a)`. This will work with both past and future versions of NumPy.

Parameters a : array_like

Array from which the diagonals are taken.

> **offset** : int, optional
>
>> Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).
>
> **axis1** : int, optional
>
>> Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
>
> **axis2** : int, optional
>
>> Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).
>
> **Returns array_of_diagonals** : ndarray
>
>> If `a` is 2-D, then a 1-D array containing the diagonal and of the same type as `a` is returned unless `a` is a `matrix`, in which case a 1-D array rather than a (2-D) `matrix` is returned in order to maintain backward compatibility.
>>
>> If `a.ndim > 2`, then the dimensions specified by `axis1` and `axis2` are removed, and a new axis inserted at the end corresponding to the diagonal.

Raises

ValueError If the dimension of `a` is less than 2.

See also:

diag() MATLAB work-a-like for 1-D and 2-D arrays.

diagflat() Create diagonal arrays.

trace() Sum along diagonals.

Examples

```
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```

A 3-D example:

```
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0,  # Main diagonals of two arrays created by skipping
...            0,  # across the outer(left)-most axis last and
...            1)  # the "middle" (row) axis first.
```

(continues on next page)

```
array([[0, 6],
       [1, 7]])
```

The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are "packed" in rows.

```
>>> a[:,:,0]  # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:,:,1]  # main diagonal is [1 7]
array([[1, 3],
       [5, 7]])
```

The anti-diagonal can be obtained by reversing the order of elements using either `numpy.flipud` or `numpy.fliplr`.

```
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.fliplr(a).diagonal()  # Horizontal flip
array([2, 4, 6])
>>> np.flipud(a).diagonal()  # Vertical flip
array([6, 4, 2])
```

Note that the order in which the diagonal is retrieved varies depending on the flip function.

dot(a, b[, out])	Dot product of two arrays.

dot function

(*Shortest import*: `from brian2 import dot`)

`brian2.units.unitsafefunctions.`**`dot`**(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both `a` and `b` are 1-D arrays, it is inner product of vectors (without complex conjugation).

- If both `a` and `b` are 2-D arrays, it is matrix multiplication, but using `matmul()` or `a @ b` is preferred.

- If either `a` or `b` is 0-D (scalar), it is equivalent to `multiply()` and using `numpy.multiply(a, b)` or `a * b` is preferred.

- If `a` is an N-D array and `b` is a 1-D array, it is a sum product over the last axis of `a` and `b`.

- If `a` is an N-D array and `b` is an M-D array (where `M>=2`), it is a sum product over the last axis of `a` and the second-to-last axis of `b`:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

Parameters **a** : array_like

First argument.

b : array_like

Second argument.

out : ndarray, optional

Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for `dot(a,b)`. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

Returns output : ndarray

Returns the dot product of `a` and `b`. If `a` and `b` are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If `out` is given, then it is returned.

Raises

ValueError If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.

See also:

vdot() Complex-conjugating dot product.

tensordot() Sum products over arbitrary axes.

einsum() Einstein summation convention.

matmul '@' operator as method with out parameter.

Examples

```
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it is the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

[*exp*](x, /[, out, where, casting, order, ...])	Calculate the exponential of all elements in the input array.

exp function

(*Shortest import*: `from brian2 import exp`)

`brian2.units.unitsafefunctions.`**`exp`**(*x*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[*, signature*, *extobj*])

Calculate the exponential of all elements in the input array.

> **Parameters** **x** : array_like
>
>> Input values.
>>
>> **out** : ndarray, None, or tuple of ndarray and None, optional
>>
>>> A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
>>
>> **where** : array_like, optional
>>
>>> This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
>>
>> ****kwargs** :
>>
>>> For other keyword-only arguments, see the ufunc docs.
>
> **Returns** **out** : ndarray or scalar
>
>> Output array, element-wise exponential of `x`. This is a scalar if `x` is a scalar.

See also:

`expm1()` Calculate `exp(x) - 1` for all elements in the array.

`exp2` Calculate `2**x` for all elements in the array.

Notes

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

References

[R19], [R20]

Examples

Plot the magnitude and phase of `exp(x)` in the complex plane:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='gray')
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='hsv')
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```

linspace(start, stop[, num, endpoint, ...])	Return evenly spaced numbers over a specified interval.

linspace function

(*Shortest import*: `from brian2 import linspace`)

`brian2.units.unitsafefunctions.`**`linspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *axis=0*)

Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the interval [`start`, *stop()*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar `start` and *stop()* are now supported.

> **Parameters** **start** : array_like
>
>> The starting value of the sequence.
>
> **stop** : array_like
>
>> The end value of the sequence, unless `endpoint` is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop()* is excluded. Note that the step size changes when `endpoint` is False.
>
> **num** : int, optional
>
>> Number of samples to generate. Default is 50. Must be non-negative.
>
> **endpoint** : bool, optional
>
>> If True, *stop()* is the last sample. Otherwise, it is not included. Default is True.
>
> **retstep** : bool, optional
>
>> If True, return (`samples`, `step()`), where `step()` is the spacing between samples.

> **dtype** : dtype, optional
>
>> The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.
>>
>> New in version 1.9.0.
>
> **axis** : int, optional
>
>> The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.
>>
>> New in version 1.16.0.
>
> **Returns samples** : ndarray
>
>> There are `num` equally spaced samples in the closed interval `[start,  stop]` or the half-open interval `[start,  stop)` (depending on whether `endpoint` is True or False).
>
> **step** : float, optional
>
>> Only returned if `retstep` is True
>>
>> Size of spacing between samples.

See also:

arange() Similar to *linspace()*, but uses a step size (instead of the number of samples).

geomspace() Similar to *linspace()*, but with numbers spaced evenly on a log scale (a geometric progression).

logspace() Similar to geomspace(), but with the end points specified as logarithms.

Examples

```
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

log(x, /[, out, where, casting, order, ...])	Natural logarithm, element-wise.

log function

(*Shortest import*: `from brian2 import log`)

`brian2.units.unitsafefunctions.`**`log`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

Natural logarithm, element-wise.

The natural logarithm *log()* is the inverse of the exponential function, so that `log(exp(x)) = x`. The natural logarithm is logarithm in base `e`.

Parameters x : array_like

> Input value.

out : ndarray, None, or tuple of ndarray and None, optional

> A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

> This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

> For other keyword-only arguments, see the ufunc docs.

Returns y : ndarray

> The natural logarithm of `x`, element-wise. This is a scalar if `x` is a scalar.

See also:

`log10()`, `log2`, `log1p()`, `emath.log`

Notes

Logarithm is a multivalued function: for each `x` there is an infinite number of `z` such that `exp(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi, pi]`.

For real-valued input data types, *log()* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, *log()* is a complex analytical function that has a branch cut `[-inf, 0]` and is continuous from above on it. *log()* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[R21], [R22]

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

---

ravel(a[, order])	Return a contiguous flattened array.

---

ravel function

(*Shortest import*: `from brian2 import ravel`)

`brian2.units.unitsafefunctions.`**`ravel`**(*a*, *order='C'*)

    Return a contiguous flattened array.

    A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

    As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

    Parameters a : array_like

        Input array. The elements in `a` are read in the order specified by `order`, and packed as a 1-D array.

    order : {'C','F', 'A', 'K'}, optional

        The elements of `a` are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if `a` is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

    Returns y : array_like

        y is an array of the same subtype as `a`, with shape `(a.size,)`. Note that matrices are special cased for backward compatibility, if `a` is a matrix, then y is a 1-D ndarray.

    See also:

`ndarray.flat` 1-D iterator over an array.

`ndarray.flatten` 1-D array copy of the elements of an array in row-major order.

`ndarray.reshape` Change the shape of an array without changing its data.

Notes

In row-major, C-style order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for column-major, Fortran-style index ordering.

When a view is desired in as many cases as possible, `arr.reshape(-1)` may be preferable.

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ravel(x)
array([1, 2, 3, 4, 5, 6])
```

```
>>> x.reshape(-1)
array([1, 2, 3, 4, 5, 6])
```

```
>>> np.ravel(x, order='F')
array([1, 4, 2, 5, 3, 6])
```

When `order` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> np.ravel(x.T)
array([1, 4, 2, 5, 3, 6])
>>> np.ravel(x.T, order='A')
array([1, 2, 3, 4, 5, 6])
```

When `order` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

sin(x, /[, out, where, casting, order, ...])	Trigonometric sine, element-wise.

sin function

(*Shortest import*: `from brian2 import sin`)

`brian2.units.unitsafefunctions.`**`sin`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*])

Trigonometric sine, element-wise.

> **Parameters x** : array_like
>
>> Angle, in radians (2π rad equals 360 degrees).
>
> **out** : ndarray, None, or tuple of ndarray and None, optional
>
>> A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
>
> **where** : array_like, optional
>
>> This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.
>
> ****kwargs** :
>
>> For other keyword-only arguments, see the ufunc docs.
>
> **Returns y** : array_like
>
>> The sine of each element of x. This is a scalar if x is a scalar.

See also:

arcsin(), *sinh()*, *cos()*

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

sinh(x, /[, out, where, casting, order, ...])	Hyperbolic sine, element-wise.

sinh function

(*Shortest import*: `from brian2 import sinh`)

`brian2.units.unitsafefunctions.`**`sinh`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*])

Hyperbolic sine, element-wise.

Equivalent to `1/2 * (np.exp(x) - np.exp(-x))` or `-1j * np.sin(1j*x)`.

Parameters x : array_like

Input array.

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns y : ndarray

The corresponding hyperbolic sine values. This is a scalar if `x` is a scalar.

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

Examples

```
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j) # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

tan(x, /[, out, where, casting, order, ...])	Compute tangent element-wise.

tan function

(*Shortest import*: `from brian2 import tan`)

`brian2.units.unitsafefunctions.`**`tan`**(*x*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*])

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

Parameters **x** : array_like

Input array.

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs** :

> For other keyword-only arguments, see the ufunc docs.

Returns **y** : ndarray

> The corresponding tangent values. This is a scalar if `x` is a scalar.

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`tanh`(x, /[, out, where, casting, order, ...])	Compute hyperbolic tangent element-wise.

tanh function

(*Shortest import*: `from brian2 import tanh`)

`brian2.units.unitsafefunctions.`**`tanh`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*])

> Compute hyperbolic tangent element-wise.
>
> Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.
>
> **Parameters** **x** : array_like
>
> > Input array.
>
> **out** : ndarray, None, or tuple of ndarray and None, optional
>
> > A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value. Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs :

For other keyword-only arguments, see the ufunc docs.

Returns y : ndarray

The corresponding hyperbolic tangent values. This is a scalar if `x` is a scalar.

Notes

If `out` is provided, the function writes the result into it, and returns a reference to `out`. (See Examples)

References

[R23], [R24]

Examples

```
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`trace`(a[, offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.

trace function

(*Shortest import*: `from brian2 import trace`)

`brian2.units.unitsafefunctions.`**`trace`**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Return the sum along diagonals of the array.

If `a` is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements `a[i,i+offset]` for all i.

If `a` has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of `a` with `axis1` and `axis2` removed.

> **Parameters a** : array_like
>
>> Input array, from which the diagonals are taken.
>
>> **offset** : int, optional
>
>> Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to 0.
>
>> **axis1, axis2** : int, optional
>
>> Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of `a`.
>
>> **dtype** : dtype, optional
>
>> Determines the data-type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and `a` is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of `a`.
>
>> **out** : ndarray, optional
>
>> Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output.
>
> **Returns sum_along_diagonals** : ndarray
>
>> If `a` is 2-D, the sum along the diagonal is returned. If `a` has larger dimensions, then an array of sums along diagonals is returned.

> **See also:**
>
> `diag()`, *`diagonal()`*, `diagflat()`

Examples

```
>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])
```

```
>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)
```

| *where*(condition, [x, y]) | Return elements chosen from x or y depending on con- |
| | dition. |

where function

(*Shortest import*: `from brian2 import where`)

`brian2.units.unitsafefunctions.`**`where`**(*condition*$[, x, y]$)

Return elements chosen from x or y depending on `condition`.

Note: When only `condition` is provided, this function is a shorthand for `np.asarray(condition).nonzero()`. Using `nonzero()` directly should be preferred, as it behaves correctly for subclasses. The rest of this documentation covers only the case where all three arguments are provided.

> **Parameters condition** : array_like, bool
>
> > Where True, yield x, otherwise yield y.
>
> **x, y** : array_like
>
> > Values from which to choose. x, y and `condition` need to be broadcastable to some shape.
>
> **Returns out** : ndarray
>
> > An array with elements from x where `condition` is True, and elements from y elsewhere.

> **See also:**
>
> `choose()`
>
> **`nonzero()`** The function that is called when x and y are omitted

Notes

If all the arrays are 1-D, *where()* is equivalent to:

```
[xv if c else yv
 for c, xv, yv in zip(condition, x, y)]
```

Examples

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.where(a < 5, a, 10*a)
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

This can be used on multidimensional arrays too:

```
>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

The shapes of x, y, and the condition are broadcast together:

```
>>> x, y = np.ogrid[:3, :4]
>>> np.where(x < y, x, 10 + y)  # both x and 10+y are broadcast
array([[10,  0,  0,  0],
       [10, 11,  1,  1],
       [10, 11, 12,  2]])
```

```
>>> a = np.array([[0, 1, 2],
...               [0, 2, 4],
...               [0, 3, 6]])
>>> np.where(a < 4, a, -1)  # -1 is broadcast
array([[ 0,  1,  2],
       [ 0,  2, -1],
       [ 0,  3, -1]])
```

wrap_function_to_method(func)	Wraps a function so that it calls the corresponding method on the Quantities object (if called with a Quantities object as the first argument).

wrap_function_to_method function

(*Shortest import*: `from brian2.units.unitsafefunctions import wrap_function_to_method`)

brian2.units.unitsafefunctions.**wrap_function_to_method**(*func*)

Wraps a function so that it calls the corresponding method on the Quantities object (if called with a Quantities object as the first argument). All other arguments are left untouched.

6.7.16 utils package

Utility functions for Brian.

Exported members: *get_logger()*, *BrianLogger*, *std_silent*

arrays module

Helper module containing functions that operate on numpy arrays.

Functions

calc_repeats(delay)	Calculates offsets corresponding to an array, where repeated values are subsequently numbered, i.e. if there n identical values, the returned array will have values from 0 to n-1 at their positions.

calc_repeats function

(*Shortest import*: `from brian2.utils.arrays import calc_repeats`)

`brian2.utils.arrays.`**`calc_repeats`**(*delay*)

> Calculates offsets corresponding to an array, where repeated values are subsequently numbered, i.e. if there n identical values, the returned array will have values from 0 to n-1 at their positions. The code is complex because tricks are needed for vectorisation.
>
> This function is used in the Python `SpikeQueue` to calculate the offset array for the insertion of spikes with their respective delays into the queue and in the numpy code for synapse creation to calculate how many synapses for each source-target pair exist.

Examples

```
>>> import numpy as np
>>> print(calc_repeats(np.array([7, 5, 7, 3, 7, 5])))
[0 0 1 0 2 1]
```

`caching` module

Module to support caching of function results to memory (used to cache results of parsing, generation of state update code, etc.). Provides the *cached* decorator.

Classes

CacheKey()	Mixin class for objects that will be used as keys for caching (e.g.

CacheKey class

(*Shortest import*: `from brian2.utils.caching import CacheKey`)

class `brian2.utils.caching.`**`CacheKey`**

> Bases: `object`
>
> Mixin class for objects that will be used as keys for caching (e.g. `Variable` objects) and have to define a certain "identity" with respect to caching. This "identity" is different from standard Python hashing and equality checking: a `Variable` for example would be considered "identical" for caching purposes regardless which object (e.g. *NeuronGroup*) it belongs to (because this does not matter for parsing, creating abstract code, etc.) but this of course matters for the values it refers to and therefore for comparison of equality to other variables.
>
> Classes that mix in the *CacheKey* class should re-define the `_cache_irrelevant_attributes` attribute to note all the attributes that should be ignored. The property `_state_tuple` will refer to a tuple of all attributes that were not excluded in such a way; this tuple will be used as the key for caching purposes.

Attributes

_cache_irrelevant_attributes	Set of attributes that should not be considered for caching of state update code, etc.

Details

_cache_irrelevant_attributes
Set of attributes that should not be considered for caching of state update code, etc.

Functions

cached(func)	Decorator to cache a function so that it will not be re-evaluated when called with the same arguments.

cached function

(*Shortest import*: `from brian2.utils.caching import cached`)

brian2.utils.caching.**cached**(*func*)
Decorator to cache a function so that it will not be re-evaluated when called with the same arguments. Uses the `_hashable` function to make arguments usable as a dictionary key even though they mutable (lists, dictionaries, etc.).

> **Parameters** **func** : function
>
> > The function to decorate.
>
> **Returns** **decorated** : function
>
> > The decorated function.

Notes

This is *not* a general-purpose caching decorator in any way comparable to `functools.lru_cache` or joblib's caching functions. It is very simplistic (no maximum cache size, no normalization of calls, e.g. `foo(3)` and `foo(x=3)` are not considered equivalent function calls) and makes very specific assumptions for our use case. Most importantly, `Variable` objects are considered to be identical when they refer to the same object, even though the actually stored values might have changed.

environment module

Utility functions to get information about the environment Brian is running in.

Functions

running_from_ipython()	Check whether we are currently running under ipython.

running_from_ipython function

(*Shortest import*: `from brian2.utils.environment import running_from_ipython`)

brian2.utils.environment.**running_from_ipython**()
    Check whether we are currently running under ipython.

> **Returns ipython** : bool
>
> > Whether running under ipython or not.

`filelock` module

A platform independent file lock that supports the with-statement.

Exported members: *Timeout*, *BaseFileLock*, *WindowsFileLock*, *UnixFileLock*, *SoftFileLock*, *FileLock*

Classes

BaseFileLock(lock_file[, timeout])	Implements the base class of a file lock.

BaseFileLock class

(*Shortest import*: `from brian2.utils.filelock import BaseFileLock`)

class brian2.utils.filelock.**BaseFileLock**(*lock_file*, *timeout=- 1*)
    Bases: `object`

Implements the base class of a file lock.

Attributes

is_locked	True, if the object holds the file lock.
lock_file	The path to the lock file.
timeout	You can set a default timeout for the filelock.

Methods

acquire([timeout, poll_intervall])	Acquires the file lock or fails with a *Timeout* error.
release([force])	Releases the file lock.

Details

is_locked
> True, if the object holds the file lock.

> Changed in version 2.0.0: This was previously a method and is now a property.

lock_file
> The path to the lock file.

timeout
> You can set a default timeout for the filelock. It will be used as fallback value in the acquire method, if no timeout value (*None*) is given.

> If you want to disable the timeout, set it to a negative value.

> A timeout of 0 means, that there is exactly one attempt to acquire the file lock.

> New in version 2.0.0.

acquire(*timeout=None*, *poll_intervall=0.05*)
> Acquires the file lock or fails with a *Timeout* error.

```
# You can use this method in the context manager (recommended)
with lock.acquire():
    pass

# Or use an equivalent try-finally construct:
lock.acquire()
try:
    pass
finally:
    lock.release()
```

> **Parameters**
>
> - **timeout** (*float*) – The maximum time waited for the file lock. If `timeout < 0`, there is no timeout and this method will block until the lock could be acquired. If `timeout` is None, the default *timeout* is used.
>
> - **poll_intervall** (*float*) – We check once in *poll_intervall* seconds if we can acquire the file lock.
>
> **Raises** *Timeout* – if the lock could not be acquired in *timeout* seconds.

> Changed in version 2.0.0: This method returns now a *proxy* object instead of *self*, so that it can be used in a with statement without side effects.

release(*force=False*)
> Releases the file lock.

> Please note, that the lock is only completly released, if the lock counter is 0.

> Also note, that the lock file itself is not automatically deleted.

> **Parameters force** (*bool*) – If true, the lock counter is ignored and the lock is released in every case.

FileLock	Alias for the lock, which should be used for the current platform.

FileLock class

(*Shortest import*: `from brian2.utils.filelock import FileLock`)

`brian2.utils.filelock.`**`FileLock`**
    Alias for the lock, which should be used for the current platform. On Windows, this is an alias for *Windows-FileLock*, on Unix for *UnixFileLock* and otherwise for *SoftFileLock*.

SoftFileLock(lock_file[, timeout])	Simply watches the existence of the lock file.

SoftFileLock class

(*Shortest import*: `from brian2.utils.filelock import SoftFileLock`)

`class` `brian2.utils.filelock.`**`SoftFileLock`**(*lock_file*, *timeout=- 1*)
    Bases: *brian2.utils.filelock.BaseFileLock*

    Simply watches the existence of the lock file.

Timeout(lock_file)	Raised when the lock could not be acquired in *timeout* seconds.

Timeout class

(*Shortest import*: `from brian2.utils.filelock import Timeout`)

`class` `brian2.utils.filelock.`**`Timeout`**(*lock_file*)
    Bases: `TimeoutError`

    Raised when the lock could not be acquired in *timeout* seconds.

Attributes

lock_file	The path of the file lock.

Details

`lock_file`
    The path of the file lock.

UnixFileLock(lock_file[, timeout])	Uses the `fcntl.flock()` to hard lock the lock file on unix systems.

UnixFileLock class

(*Shortest import*: `from brian2.utils.filelock import UnixFileLock`)

class brian2.utils.filelock.**UnixFileLock**(*lock_file*, *timeout=- 1*)
    Bases: *brian2.utils.filelock.BaseFileLock*

    Uses the `fcntl.flock()` to hard lock the lock file on unix systems.

WindowsFileLock(lock_file[, timeout])	Uses the `msvcrt.locking()` function to hard lock the lock file on windows systems.

WindowsFileLock class

(*Shortest import*: `from brian2.utils.filelock import WindowsFileLock`)

class brian2.utils.filelock.**WindowsFileLock**(*lock_file*, *timeout=- 1*)
    Bases: *brian2.utils.filelock.BaseFileLock*

    Uses the `msvcrt.locking()` function to hard lock the lock file on windows systems.

Functions

logger()	Returns the logger instance used in this module.

logger function

(*Shortest import*: `from brian2.utils.filelock import logger`)

brian2.utils.filelock.**logger**()
    Returns the logger instance used in this module.

`filetools` **module**

File system tools

Exported members: *ensure_directory*, *ensure_directory_of_file*, *in_directory*, *copy_directory*

Classes

in_directory(new_dir)	Safely temporarily work in a subdirectory

in_directory class

(*Shortest import*: `from brian2.utils.filetools import in_directory`)

class `brian2.utils.filetools.`**`in_directory`**(*new_dir*)
    Bases: `object`

    Safely temporarily work in a subdirectory

    Usage:

```
with in_directory(directory):
    ... do stuff here
```

    Guarantees that the code in the with block will be executed in directory, and that after the block is completed we return to the original directory.

Functions

copy_directory(source, target)	Copies directory source to target.

copy_directory function

(*Shortest import*: `from brian2.utils.filetools import copy_directory`)

`brian2.utils.filetools.`**`copy_directory`**(*source*, *target*)
    Copies directory source to target.

ensure_directory(d)	Ensures that a given directory exists (creates it if necessary)

ensure_directory function

(*Shortest import*: `from brian2.utils.filetools import ensure_directory`)

`brian2.utils.filetools.`**`ensure_directory`**(*d*)
    Ensures that a given directory exists (creates it if necessary)

ensure_directory_of_file(f)	Ensures that a directory exists for filename to go in (creates if necessary), and returns the directory path.

ensure_directory_of_file function

(*Shortest import*: `from brian2.utils.filetools import ensure_directory_of_file`)

`brian2.utils.filetools.`**`ensure_directory_of_file`**(*f*)
> Ensures that a directory exists for filename to go in (creates if necessary), and returns the directory path.

`logger` module

Brian's logging module.

Preferences

Logging system preferences

`logging.console_log_level = 'INFO'`

> What log level to use for the log written to the console.
>
> Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.delete_log_on_exit = True`

> Whether to delete the log and script file on exit.
>
> If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occurred. If set to `False`, all log files will be kept.

`logging.display_brian_error_message = True`

> Whether to display a text for uncaught errors, mentioning the location of the log file, the mailing list and the github issues.
>
> Defaults to `True`.

`logging.file_log = True`

> Whether to log to a file or not.
>
> If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level = 'DIAGNOSTIC'`

> What log level to use for the log written to the log file.
>
> In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.file_log_max_size = 10000000`

> The maximum size for the debug log before it will be rotated.
>
> If set to any value > `0`, the debug log will be rotated once this size is reached. Rotating the log means that the old debug log will be moved into a file in the same directory but with suffix `".1"` and the a new log file will be created with the same pathname as the original file. Only one backup is kept; if a file with suffix `".1"` already exists when rotating, it will be overwritten. If set to `0`, no log rotation will be applied. The default setting rotates the log file after 10MB.

`logging.save_script = True`

Whether to save a copy of the script that is run.

If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection = True`

Whether or not to redirect stdout/stderr to null at certain places.

This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. If *logging.std_redirection_to_file* is set to `True` as well, then the output is saved to a file and if an error occurs the name of this file will be printed.

`logging.std_redirection_to_file = True`

Whether to redirect stdout/stderr to a file.

If both `logging.std_redirection` and this preference are set to `True`, all standard output/error (most importantly output from the compiler) will be stored in files and if an error occurs the name of this file will be printed. If *logging.std_redirection* is `True` and this preference is `False`, then all standard output/error will be completely suppressed, i.e. neither be displayed nor stored in a file.

The value of this preference is ignore if *logging.std_redirection* is set to `False`.

Exported members: `get_logger()`, `BrianLogger`, `std_silent`

Classes

BrianLogger(name)	Convenience object for logging.

BrianLogger class

(*Shortest import*: `from brian2 import BrianLogger`)

class `brian2.utils.logger.`**`BrianLogger`**(*name*)

Bases: `object`

Convenience object for logging. Call `get_logger()` to get an instance of this class.

> **Parameters name** : str
>
>> The name used for logging, normally the name of the module.

Attributes

_log_messages	Class attribute for remembering log messages that should only be displayed once
console_handler	The `logging.StreamHandler` responsible for logging to the console
exception_occured	Class attribute to remember whether any exception occured
file_handler	The `logging.FileHandler` responsible for logging to the temporary log file
tmp_log	The name of the temporary log file (by default deleted after the run if no exception occurred), if any

Table 532 – continued from previous page

tmp_script	The name of the temporary copy of the main script file (by default deleted after the run if no exception occurred), if any

Methods

debug(msg[, name_suffix, once])	Log a debug message.
diagnostic(msg[, name_suffix, once])	Log a diagnostic message.
error(msg[, name_suffix, once])	Log an error message.
info(msg[, name_suffix, once])	Log an info message.
initialize()	Initialize Brian's logging system.
log_level_debug()	Set the log level to "debug".
log_level_diagnostic()	Set the log level to "diagnostic".
log_level_error()	Set the log level to "error".
log_level_info()	Set the log level to "info".
log_level_warn()	Set the log level to "warn".
suppress_hierarchy(name[, filter_log_file])	Suppress all log messages in a given hiearchy.
suppress_name(name[, filter_log_file])	Suppress all log messages with a given name.
warn(msg[, name_suffix, once])	Log a warn message.

Details

_log_messages

    Class attribute for remembering log messages that should only be displayed once

console_handler

    The `logging.StreamHandler` responsible for logging to the console

exception_occured

    Class attribute to remember whether any exception occured

file_handler

    The `logging.FileHandler` responsible for logging to the temporary log file

tmp_log

    The name of the temporary log file (by default deleted after the run if no exception occurred), if any

tmp_script

    The name of the temporary copy of the main script file (by default deleted after the run if no exception occurred), if any

debug(*msg*, *name_suffix=None*, *once=False*)

    Log a debug message.

        Parameters **msg** : str

            The message to log.

        name_suffix : str, optional

            A suffix to add to the name, e.g. a class or function name.

        once : bool, optional

            Whether this message should be logged only once and not repeated if sent another time.

diagnostic(*msg*, *name_suffix=None*, *once=False*)
  Log a diagnostic message.

> **Parameters** **msg** : str
>
>> The message to log.
>
>> **name_suffix** : str, optional
>
>> A suffix to add to the name, e.g. a class or function name.
>
>> **once** : bool, optional
>
>> Whether this message should be logged only once and not repeated if sent another time.

error(*msg*, *name_suffix=None*, *once=False*)
  Log an error message.

> **Parameters** **msg** : str
>
>> The message to log.
>
>> **name_suffix** : str, optional
>
>> A suffix to add to the name, e.g. a class or function name.
>
>> **once** : bool, optional
>
>> Whether this message should be logged only once and not repeated if sent another time.

info(*msg*, *name_suffix=None*, *once=False*)
  Log an info message.

> **Parameters** **msg** : str
>
>> The message to log.
>
>> **name_suffix** : str, optional
>
>> A suffix to add to the name, e.g. a class or function name.
>
>> **once** : bool, optional
>
>> Whether this message should be logged only once and not repeated if sent another time.

static initialize()
  Initialize Brian's logging system. This function will be called automatically when Brian is imported.

static log_level_debug()
  Set the log level to "debug".

static log_level_diagnostic()
  Set the log level to "diagnostic".

static log_level_error()
  Set the log level to "error".

static log_level_info()
  Set the log level to "info".

static log_level_warn()
  Set the log level to "warn".

static suppress_hierarchy(*name*, *filter_log_file=False*)
  Suppress all log messages in a given hiearchy.

> **Parameters** **name** : str

Suppress all log messages in the given `name` hierarchy. For example, specifying `'brian2'` suppresses all messages logged by Brian, specifying `'brian2.codegen'` suppresses all messages generated by the code generation modules.

filter_log_file : bool, optional

Whether to suppress the messages also in the log file. Defaults to `False` meaning that suppressed messages are not displayed on the console but are still saved to the log file.

static suppress_name(*name*, *filter_log_file=False*)

Suppress all log messages with a given name.

Parameters name : str

Suppress all log messages ending in the given `name`. For example, specifying `'resolution_conflict'` would suppress messages with names such as `brian2.equations.codestrings.CodeString.resolution_conflict` or `brian2.equations.equations.Equations.resolution_conflict`.

filter_log_file : bool, optional

Whether to suppress the messages also in the log file. Defaults to `False` meaning that suppressed messages are not displayed on the console but are still saved to the log file.

warn(*msg*, *name_suffix=None*, *once=False*)

Log a warn message.

Parameters msg : str

The message to log.

name_suffix : str, optional

A suffix to add to the name, e.g. a class or function name.

once : bool, optional

Whether this message should be logged only once and not repeated if sent another time.

Tutorials and examples using this

- Example *frompapers/Rossant_et_al_2011bis*

HierarchyFilter(name)	A class for suppressing all log messages in a subtree of the name hierarchy.

HierarchyFilter class

(*Shortest import*: `from brian2.utils.logger import HierarchyFilter`)

class brian2.utils.logger.**HierarchyFilter**(*name*)

Bases: object

A class for suppressing all log messages in a subtree of the name hierarchy. Does exactly the opposite as the `logging.Filter` class, which allows messages in a certain name hierarchy to *pass*.

Parameters name : str

The name hiearchy to suppress. See *BrianLogger.suppress_hierarchy* for details.

Methods

filter(record)	Filter out all messages in a subtree of the name hierarchy.

Details

filter(*record*)
  Filter out all messages in a subtree of the name hierarchy.

LogCapture(log_list[, log_level])	A class for capturing log warnings.

LogCapture class

(*Shortest import*: from brian2.utils.logger import LogCapture)

class brian2.utils.logger.**LogCapture**(*log_list*, *log_level=30*)
  Bases: logging.Handler

  A class for capturing log warnings. This class is used by *catch_logs* to allow testing in a similar way as with warnings.catch_warnings.

Methods

emit(record)	Do whatever it takes to actually log the specified logging record.
install()	Install this handler to catch all warnings.
uninstall()	Uninstall this handler and re-connect the previously installed handlers.

Details

emit(*record*)
  Do whatever it takes to actually log the specified logging record.

  This version is intended to be implemented by subclasses and so raises a NotImplementedError.

install()
  Install this handler to catch all warnings. Temporarily disconnect all other handlers.

uninstall()
  Uninstall this handler and re-connect the previously installed handlers.

NameFilter(name)	A class for suppressing log messages ending with a certain name.

NameFilter class

(*Shortest import*: `from brian2.utils.logger import NameFilter`)

class `brian2.utils.logger.`**`NameFilter`**(*name*)
    Bases: [`object`](#)

    A class for suppressing log messages ending with a certain name.

        Parameters name : str

            The name to suppress. See *`BrianLogger.suppress_name`* for details.

Methods

`filter`(record)	Filter out all messages ending with a certain name.

Details

`filter`(*record*)
    Filter out all messages ending with a certain name.

`catch_logs`([log_level])	A context manager for catching log messages.

catch_logs class

(*Shortest import*: `from brian2.utils.logger import catch_logs`)

class `brian2.utils.logger.`**`catch_logs`**(*log_level=30*)
    Bases: [`object`](#)

    A context manager for catching log messages. Use this for testing the messages that are logged. Defaults to catching warning/error messages and this is probably the only real use case for testing. Note that while this context manager is active, *all* log messages are suppressed. Using this context manager returns a list of (log level, name, message) tuples.

        Parameters log_level : int or str, optional

            The log level above which messages are caught.

Examples

```
>>> logger = get_logger('brian2.logtest')
>>> logger.warn('An uncaught warning')
WARNING  brian2.logtest: An uncaught warning
>>> with catch_logs() as l:
...     logger.warn('a caught warning')
...     print('l contains: %s' % l)
...
l contains: [('WARNING', 'brian2.logtest', 'a caught warning')]
```

std_silent([alwaysprint])	Context manager that temporarily silences stdout and stderr but keeps the output saved in a temporary file and writes it if an exception is raised.

std_silent class

(*Shortest import*: `from brian2 import std_silent`)

class `brian2.utils.logger.`**`std_silent`**(*alwaysprint=False*)

    Bases: `object`

    Context manager that temporarily silences stdout and stderr but keeps the output saved in a temporary file and writes it if an exception is raised.

Attributes

dest_stderr
dest_stdout

Methods

close()

Details

    dest_stderr = None

    dest_stdout = None

    classmethod close()

Functions

brian_excepthook(exc_type, exc_obj, exc_tb)	Display a message mentioning the debug log in case of an uncaught exception.

brian_excepthook function

(*Shortest import*: `from brian2.utils.logger import brian_excepthook`)

`brian2.utils.logger.`**`brian_excepthook`**(*exc_type*, *exc_obj*, *exc_tb*)
Display a message mentioning the debug log in case of an uncaught exception.

[*clean_up_logging*](#)()	Shutdown the logging system and delete the debug log file if no error occured.

clean_up_logging function

(*Shortest import*: `from brian2.utils.logger import clean_up_logging`)

`brian2.utils.logger.`**`clean_up_logging`**()
Shutdown the logging system and delete the debug log file if no error occured.

[*get_logger*](#)([module_name])	Get an object that can be used for logging.

get_logger function

(*Shortest import*: `from brian2 import get_logger`)

`brian2.utils.logger.`**`get_logger`**(*module_name='brian2'*)
Get an object that can be used for logging.

> **Parameters** **module_name** : str
>
> > The name used for logging, should normally be the module name as returned by `__name__`.
>
> **Returns** **logger** : [*BrianLogger*](#)

[*log_level_validator*](#)(log_level)	

log_level_validator function

(*Shortest import*: `from brian2.utils.logger import log_level_validator`)

`brian2.utils.logger.`**`log_level_validator`**(*log_level*)

`stringtools` module

A collection of tools for string formatting tasks.

Exported   members: *indent*,   *deindent*,   *word_substitute*,   *replace*,
get_identifiers,   *strip_empty_lines*,   *stripped_deindented_lines*,
strip_empty_leading_and_trailing_lines, *code_representation*, *SpellChecker*

Classes

SpellChecker(words[, alphabet])	A simple spell checker that will be used to suggest the correct name if the user made a typo (e.g.

SpellChecker class

(*Shortest import*: `from brian2.utils.stringtools import SpellChecker`)

class `brian2.utils.stringtools.`**`SpellChecker`**(*words*, *alpha-bet='abcdefghijklmnopqrstuvwxyz0123456789_'*)

> Bases: `object`
>
> A simple spell checker that will be used to suggest the correct name if the user made a typo (e.g. for state variable names).
>
> > **Parameters words** : iterable of str
> >
> > > The known words
> >
> > **alphabet** : iterable of str, optional
> >
> > > The allowed characters. Defaults to the characters allowed for identifiers, i.e. ascii characters, digits and the underscore.

Methods

edits1(word)	
known(words)	
known_edits2(word)	
suggest(word)	

Details

edits1(*word*)

known(*words*)

known_edits2(*word*)

suggest(*word*)

Functions

code_representation(code)	Returns a string representation for several different formats of code

code_representation function

(*Shortest import*: `from brian2.utils.stringtools import code_representation`)

`brian2.utils.stringtools.`**`code_representation`**(*code*)
    Returns a string representation for several different formats of code

    Formats covered include: - A single string - A list of statements/strings - A dict of strings - A dict of lists of statements/strings

deindent(text[, numtabs, spacespertab, ...])	Returns a copy of the string with the common indentation removed.

deindent function

(*Shortest import*: `from brian2.utils.stringtools import deindent`)

`brian2.utils.stringtools.`**`deindent`**(*text*, *numtabs=None*, *spacespertab=4*, *docstring=False*)
    Returns a copy of the string with the common indentation removed.

    Note that all tab characters are replaced with `spacespertab` spaces.

    If the `docstring` flag is set, the first line is treated differently and is assumed to be already correctly tabulated.

    If the `numtabs` option is given, the amount of indentation to remove is given explicitly and not the common indentation.

Examples

Normal strings, e.g. function definitions:

```
>>> multiline = '''    def f(x):
...           return x**2'''
>>> print(multiline)
    def f(x):
        return x**2
>>> print(deindent(multiline))
def f(x):
    return x**2
>>> print(deindent(multiline, docstring=True))
```

(continues on next page)

```
    def f(x):
return x**2
>>> print(deindent(multiline, numtabs=1, spacespertab=2))
  def f(x):
        return x**2
```

Docstrings:

```
>>> docstring = '''First docstring line.
...       This line determines the indentation.'''
>>> print(docstring)
First docstring line.
    This line determines the indentation.
>>> print(deindent(docstring, docstring=True))
First docstring line.
This line determines the indentation.
```

get_identifiers(expr[, include_numbers])	Return all the identifiers in a given string `expr`, that is everything that matches a programming language variable like expression, which is here implemented as the regexp `\b[A-Za-z_][A-Za-z0-9_]*\b`.

get_identifiers function

(*Shortest import*: `from brian2.utils.stringtools import get_identifiers`)

`brian2.utils.stringtools.`**`get_identifiers`**(*expr*, *include_numbers=False*)

Return all the identifiers in a given string `expr`, that is everything that matches a programming language variable like expression, which is here implemented as the regexp `\b[A-Za-z_][A-Za-z0-9_]*\b`.

> **Parameters expr** : str
>
>> The string to analyze
>
> **include_numbers** : bool, optional
>
>> Whether to include number literals in the output. Defaults to `False`.
>
> **Returns identifiers** : set
>
>> A set of all the identifiers (and, optionally, numbers) in `expr`.

Examples

```
>>> expr = '3-a*_b+c5+8+f(A - .3e-10, tau_2)*17'
>>> ids = get_identifiers(expr)
>>> print(sorted(list(ids)))
['A', '_b', 'a', 'c5', 'f', 'tau_2']
>>> ids = get_identifiers(expr, include_numbers=True)
>>> print(sorted(list(ids)))
['.3e-10', '17', '3', '8', 'A', '_b', 'a', 'c5', 'f', 'tau_2']
```

indent(text[, numtabs, spacespertab, tab])	Indents a given multiline string.

indent function

(*Shortest import*: `from brian2.utils.stringtools import indent`)

`brian2.utils.stringtools.`**`indent`**(*text*, *numtabs=1*, *spacespertab=4*, *tab=None*)

Indents a given multiline string.

By default, indentation is done using spaces rather than tab characters. To use tab characters, specify the tab character explictly, e.g.:

```
indent(text, tab='        ')
```

Note that in this case `spacespertab` is ignored.

Examples

```
>>> multiline = '''def f(x):
...     return x*x'''
>>> print(multiline)
def f(x):
    return x*x
>>> print(indent(multiline))
    def f(x):
        return x*x
>>> print(indent(multiline, numtabs=2))
        def f(x):
            return x*x
>>> print(indent(multiline, spacespertab=2))
  def f(x):
      return x*x
>>> print(indent(multiline, tab='####'))
####def f(x):
####    return x*x
```

[*replace*](s, substitutions)	Applies a dictionary of substitutions.

replace function

(*Shortest import*: `from brian2.utils.stringtools import replace`)

`brian2.utils.stringtools.`**`replace`**(*s*, *substitutions*)

Applies a dictionary of substitutions. Simpler than [*word_substitute*](), it does not attempt to only replace words

[*strip_empty_leading_and_trailing_lines*](s)	Removes all empty leading and trailing lines in the multiline string `s`.

strip_empty_leading_and_trailing_lines function

(*Shortest import*: `from brian2.utils.stringtools import strip_empty_leading_and_trailing_lines`)

`brian2.utils.stringtools.`**`strip_empty_leading_and_trailing_lines`**(*s*)
    Removes all empty leading and trailing lines in the multi-line string `s`.

strip_empty_lines(s)	Removes all empty lines from the multi-line string `s`.

strip_empty_lines function

(*Shortest import*: `from brian2.utils.stringtools import strip_empty_lines`)

`brian2.utils.stringtools.`**`strip_empty_lines`**(*s*)
    Removes all empty lines from the multi-line string `s`.

Examples

```
>>> multiline = '''A string with
...
... an empty line.'''
>>> print(strip_empty_lines(multiline))
A string with
an empty line.
```

stripped_deindented_lines(code)	Returns a list of the lines in a multi-line string, deindented.

stripped_deindented_lines function

(*Shortest import*: `from brian2.utils.stringtools import stripped_deindented_lines`)

`brian2.utils.stringtools.`**`stripped_deindented_lines`**(*code*)
    Returns a list of the lines in a multi-line string, deindented.

word_substitute(expr, substitutions)	Applies a dict of word substitutions.

word_substitute function

(*Shortest import*: `from brian2.utils.stringtools import word_substitute`)

`brian2.utils.stringtools.`**`word_substitute`**(*expr*, *substitutions*)
    Applies a dict of word substitutions.

    The dict `substitutions` consists of pairs `(word, rep)` where each word `word` appearing in `expr` is replaced by `rep`. Here a 'word' means anything matching the regexp `\bword\b`.

Examples

```
>>> expr = 'a*_b+c5+8+f(A)'
>>> print(word_substitute(expr, {'a':'banana', 'f':'func'}))
banana*_b+c5+8+func(A)
```

`topsort` **module**

Exported members: `topsort`

Functions

`topsort`(graph)	Topologically sort a graph

topsort function

(*Shortest import*: `from brian2.utils.topsort import topsort`)

brian2.utils.topsort.**topsort**(*graph*)

Topologically sort a graph

The graph should be of the form `{node: [list of nodes], ...}`.

DEVELOPER'S GUIDE

This section is intended as a guide to how Brian functions internally for people developing Brian itself, or extensions to Brian. It may also be of some interest to others wishing to better understand how Brian works internally.

7.1 Coding guidelines

The basic principles of developing Brian are:

1. For the user, the emphasis is on making the package flexible, readable and easy to use. See the paper "The Brian simulator" in Frontiers in Neuroscience for more details.

2. For the developer, the emphasis is on keeping the package maintainable by a small number of people. To this end, we use stable, well maintained, existing open source packages whenever possible, rather than writing our own code.

7.1.1 Development workflow

Brian development is done in a git repository on github. Continuous integration testing is provided by travis CI, code coverage is measured with coveralls.io.

The repository structure

Brian's repository structure is very simple, as we are normally not supporting older versions with bugfixes or other complicated things. The *master* branch of the repository is the basis for releases, a release is nothing more than adding a tag to the branch, creating the tarball, etc. The *master* branch should always be in a deployable state, i.e. one should be able to use it as the base for everyday work without worrying about random breakages due to updates. To ensure this, no commit ever goes into the *master* branch without passing the test suite before (see below). The only exception to this rule is if a commit not touches any code files, e.g. additions to the README file or to the documentation (but even in this case, care should be taken that the documentation is still built correctly).

For every feature that a developer works on, a new branch should be opened (normally based on the *master* branch), with a descriptive name (e.g. `add-numba-support`). For developers that are members of "brian-team", the branch should ideally be created in the main repository. This way, one can easily get an overview over what the "core team" is currently working on. Developers who are not members of the team should fork the repository and work in their own repository (if working on multiple issues/features, also using branches).

Implementing a feature/fixing a bug

Every new feature or bug fix should be done in a dedicated branch and have an issue in the issue database. For bugs, it is important to not only fix the bug but also to introduce a new test case (see *Testing*) that makes sure that the bug will not ever be reintroduced by other changes. It is often a good idea to first define the test cases (that should fail) and then work on the fix so that the tests pass. As soon as the feature/fix is complete *or* as soon as specific feedback on the code is needed, open a "pull request" to merge the changes from your branch into *master*. In this pull request, others can comment on the code and make suggestions for improvements. New commits to the respective branch automatically appear in the pull request which makes it a great tool for iterative code review. Even more useful, travis will automatically run the test suite on the result of the merge. As a reviewer, always wait for the result of this test (it can take up to 30 minutes or so until it appears) before doing the merge and never merge when a test fails. As soon as the reviewer (someone from the core team and not the author of the feature/fix) decides that the branch is ready to merge, he/she can merge the pull request and optionally delete the corresponding branch (but it will be hidden by default, anyway).

Useful links

- The Brian repository: https://github.com/brian-team/brian2
- Travis testing for Brian: https://travis-ci.org/brian-team/brian2
- Code Coverage for Brian: https://coveralls.io/github/brian-team/brian2
- The Pro Git book: https://git-scm.com/book/en/v2
- github's documentation on pull requests: https://help.github.com/articles/using-pull-requests

7.1.2 Coding conventions

General recommendations

Syntax is chosen as much as possible from the user point of view, to reflect the concepts as directly as possible. Ideally, a Brian script should be readable by someone who doesn't know Python or Brian, although this isn't always possible. Function, class and keyword argument names should be explicit rather than abbreviated and consistent across Brian. See Romain's paper On the design of script languages for neural simulators for a discussion.

We use the PEP-8 coding conventions for our code. This in particular includes the following conventions:

- Use 4 spaces instead of tabs per indentation level
- Use spaces after commas and around the following binary operators: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- Do *not* use spaces around the equals sign in keyword arguments or when specifying default values. Neither put spaces immediately inside parentheses, brackets or braces, immediately before the open parenthesis that starts the argument list of a function call, or immediately before the open parenthesis that starts an indexing or slicing.
- Avoid using a backslash for continuing lines whenever possible, instead use Python's implicit line joining inside parentheses, brackets and braces.
- imports should be on different lines (e.g. do not use `import sys, os`) and should be grouped in the following order, using blank lines between each group:
  1. standard library imports
  2. third-party library imports (e.g. numpy, scipy, sympy, …)
  3. brian imports

- Use absolute imports for everything outside of "your" package, e.g. if you are working in `brian2.equations`, import functions from the stringtools modules via `from brian2.utils.stringtools import ..` `..` Use the full path when importing, e.g. do `from brian2.units.fundamentalunits import seconds` instead of `from brian2 import seconds`.

- Use "new-style" relative imports for everything in "your" package, e.g. in `brian2.codegen.functions.py` import the `Function` class as `from .specifiers import Function`.

- Do not use wildcard imports (`from brian2 import *`), instead import only the identifiers you need, e.g. `from brian2 import NeuronGroup, Synapses`. For packages like numpy that are used a lot, use `import numpy as np`. But note that the user should still be able to do something like `from brian2 import *` (and this style can also be freely used in examples and tests, for example). Modules always have to use the `__all__` mechanism to specify what is being made available with a wildcard import. As an exception from this rule, the main `brian2/__init__.py` may use wildcard imports.

String formatting

In general, we use Python f-strings instead of the `.format` method or the `%` operator to format strings. For example, rather use:

```python
raise KeyError(f"Unknown variable '{var}'")  # ✓
```

instead of:

```python
raise KeyError("Unknown variable '{}'".format(var))  # ⍰
raise KeyError("Unknown variable %s" % var)  # ⍰
```

There are some corner cases where it still makes sense to use either of these, though. The `format` method can be useful when processing several strings instead of single literals:

```python
formatted = []
for s in strings:
    formatted.append(s.format(**values))
```

The `%` operator, or string concatenation, can be used when dealing with strings that contain curly braces, which would become difficult to read as an f-string:

```python
latex_code = r'\begin{equation}%s\end{equation}' % equation  # OK
latex_code = r'\begin{equation}' + equation + r'\end{equation}' # OK
```

Python does not make a difference between single quotation marks and double quotation marks. For consistency, try to follow the following rules:

- docstrings should always be enclosed in triple double quotes, following PEP 257.

- User-facing text (e.g. error messages) should use double quotes, and single quotes for marking words within the string. Example: `"Missing 'threshold' argument"`

- General strings with internal meaning (e.g. dictionary keys) should use single quotation marks. Example: `events['spike']`

- Use your own judgement for other strings, e.g. generated code. If you need to use single or double quotes *within* the string, use the other quote type to avoid having to resort to backslashes. Example: `include = f'#include "{header_file}"'`

Commits only changing the style

Please do not make commits that only change the code style in a file, even though many files do not completely follow the rules mentioned earlier. However, if you are commiting edits to a file for different reasons, please do follow this style for your changes and, if necessary, change the surrounding code to fit the style (within reason).

We sometimes do make big commits updating the style in our code, which can make using tools like `git blame` more difficult, since many lines affected by such commits. We add the references to such commits to a file `.git-blame-ignore-revs` in the main directory, and you can tell `git blame` to ignore these commits with:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

7.1.3 Representing Brian objects

`__repr__` and `__str__`

Every class should specify or inherit useful `__repr__` and `__str__` methods. The `__repr__` method should give the "official" representation of the object; if possible, this should be a valid Python expression, ideally allowing for `eval(repr(x)) == x`. The `__str__` method on the other hand, gives an "informal" representation of the object. This can be anything that is helpful but does not have to be Python code. For example:

```
>>> import numpy as np
>>> ar = np.array([1, 2, 3]) * mV
>>> print(ar)   # uses __str__
[ 1.  2.  3.] mV
>>> ar   # uses __repr__
array([ 1.,  2.,  3.]) * mvolt
```

If the representation returned by `__repr__` is not Python code, it should be enclosed in `<...>`, e.g. a *Synapses* representation might be `<Synapses object with 64 synapses>`.

If you don't want to make the distinction between `__repr__` and `__str__`, simply define only a `__repr__` function, it will be used instead of `__str__` automatically (no need to write `__str__` = `__repr__`). Finally, if you include the class name in the representation (which you should in most cases), use `self.__class__.__name__` instead of spelling out the name explicitly – this way it will automatically work correctly for subclasses. It will also prevent you from forgetting to update the class name in the representation if you decide to rename the class.

LaTeX representations with sympy

Brian objects dealing with mathematical expressions and equations often internally use sympy. Sympy's `latex` function does a nice job of converting expressions into LaTeX code, using fractions, root symbols, etc. as well as converting greek variable names into corresponding symbols and handling sub- and superscripts. For the conversion of variable names to work, they should use an underscore for subscripts and two underscores for superscripts:

```
>>> from sympy import latex, Symbol
>>> tau_1__e = Symbol('tau_1__e')
>>> print(latex(tau_1__e))
\tau^{e}_{1}
```

Sympy's printer supports formatting arbitrary objects, all they have to do is to implement a `_latex` method (no trailing underscore). For most Brian objects, this is unnecessary as they will never be formatted with sympy's LaTeX printer. For some core objects, in particular the units, is is useful, however, as it can be reused in LaTeX representations for ipython (see below). Note that the `_latex` method should not return `$` or `\begin{equation}` (sympy's method includes a `mode` argument that wraps the output automatically).

Representations for ipython

"Old" ipython console

In particular for representations involing arrays or lists, it can be useful to break up the representation into chunks, or indent parts of the representation. This is supported by the ipython console's "pretty printer". To make this work for a class, add a _repr_pretty_(self, p, cycle) (note the *single* underscores) method. You can find more information in the ipython documentation .

"New" ipython console (qtconsole and notebook)

The new ipython consoles, the qtconsole and the ipython notebook support a much richer set of representations for objects. As Brian deals a lot with mathematical objects, in particular the LaTeX and to a lesser extent the HTML formatting capabilities of the ipython notebook are interesting. To support LaTeX representation, implement a _repr_latex_ method returning the LaTeX code (*including* $, \begin{equation} or similar). If the object already has a _latex method (see *LaTeX representations with sympy* above), this can be as simple as:

```python
def _repr_latex_(self):
    return sympy.latex(self, mode='inline')  # wraps the expression in $ .. $
```

The LaTeX rendering only supports a single mathematical block. For complex objects, e.g. *NeuronGroup* it might be useful to have a richer representation. This can be achieved by returning HTML code from _repr_html_ – this HTML code is processed by MathJax so it can include literal LaTeX code that will be transformed before it is rendered as HTML. An object containing two equations could therefore be represented with a method like this:

```python
def _repr_html_(self):
    return '''
    <h3> Equation 1 </h3>
    {eq_1}
    <h3> Equation 2 </h3>
    {eq_2}'''.format(eq_1=sympy.latex(self.eq_1, mode='equation'),
                     eq_2=sympy.latex(self.eq_2, mode='equation'))
```

7.1.4 Defensive programming

One idea for Brian 2 is to make it so that it's more likely that errors are raised rather than silently causing weird bugs. Some ideas in this line:

Synapses.source should be stored internally as a weakref Synapses._source, and Synapses.source should be a computed attribute that dereferences this weakref. Like this, if the source object isn't kept by the user, Synapses won't store a reference to it, and so won't stop it from being deallocated.

We should write an automated test that takes a piece of correct code like:

```python
NeuronGroup(N, eqs, reset='V>Vt')
```

and tries replacing all arguments by nonsense arguments, it should always raise an error in this case (forcing us to write code to validate the inputs). For example, you could create a new NonsenseObject class, and do this:

```python
nonsense = NonsenseObject()
NeuronGroup(nonsense, eqs, reset='V>Vt')
NeuronGroup(N, nonsense, reset='V>Vt')
NeuronGroup(N, eqs, nonsense)
```

In general, the idea should be to make it hard for something incorrect to run without raising an error, preferably at the point where the user makes the error and not in some obscure way several lines later.

The preferred way to validate inputs is one that handles types in a Pythonic way. For example, instead of doing something like:

```python
if not isinstance(arg, (float, int)):
    raise TypeError(...)
```

Do something like:

```python
arg = float(arg)
```

(or use try/except to raise a more specific error). In contrast to the `isinstance` check it does not make any assumptions about the type except for its ability to be converted to a float.

This approach is particular useful for numpy arrays:

```python
arr = np.asarray(arg)
```

(or `np.asanyarray` if you want to allow for array subclasses like arrays with units or masked arrays). This approach has also the nice advantage that it allows all "array-like" arguments, e.g. a list of numbers.

7.1.5 Documentation

It is very important to maintain documentation. We use the Sphinx documentation generator tools. The documentation is all hand written. Sphinx source files are stored in the `docs_sphinx` folder. The HTML files can be generated via the script `dev/tools/docs/build_html_brian2.py` and end up in the `docs` folder.

Most of the documentation is stored directly in the Sphinx source text files, but reference documentation for important Brian classes and functions are kept in the documentation strings of those classes themselves. This is automatically pulled from these classes for the reference manual section of the documentation. The idea is to keep the definitive reference documentation near the code that it documents, serving as both a comment for the code itself, and to keep the documentation up to date with the code.

The reference documentation includes all classes, functions and other objects that are defined in the modules and only documents them in the module where they were defined. This makes it possible to document a class like *Quantity* only in *brian2.units.fundamentalunits* and not additionally in *brian2.units* and `brian2`. This mechanism relies on the `__module__` attribute, in some cases, in particular when wrapping a function with a decorator (e.g. *check_units*), this attribute has to be set manually:

```python
foo.__module__ = __name__
```

Without this manual setting, the function might not be documented at all or in the wrong module.

In addition to the reference, all the examples in the examples folder are automatically included in the documentation.

Note that you can directly link to github issues using `:issue:`issue number``, e.g. writing `:issue:`33`` links to a github issue about running benchmarks for Brian 2: #33. This feature should rarely be used in the main documentation, reserve its use for release notes and important known bugs.

Docstrings

Every module, class, method or function has to start with a docstring, unless it is a private or special method (i.e. starting with _ or __) *and* it is obvious what it does. For example, there is normally no need to document `__str__` with "Return a string representation.".

For the docstring format, we use the our own sphinx extension (in `brian2/sphinxext`) based on numpydoc, allowing to write docstrings that are well readable both in sourcecode as well as in the rendered HTML. We generally follow the format used by numpy

When the docstring uses variable, class or function names, these should be enclosed in single backticks. Class and function/method names will be automatically linked to the corresponding documentation. For classes imported in the main brian2 package, you do not have to add the package name, e.g. writing `` `NeuronGroup` `` is enough. For other classes, you have to give the full path, e.g. `` `brian2.units.fundamentalunits.UnitRegistry` ``. If it is clear from the context where the class is (e.g. within the documentation of *UnitRegistry*), consider using the ~ abbreviation: `` `~brian2.units.fundamentalunits.UnitRegistry` `` displays only the class name: *UnitRegistry*. Note that you do not have to enclose the exception name in a "Raises" or "Warns" section, or the class/method/function name in a "See Also" section in backticks, they will be automatically linked (putting backticks there will lead to incorrect display or an error message),

Inline source fragments should be enclosed in double backticks.

Class docstrings follow the same conventions as method docstrings and should document the `__init__` method, the `__init__` method itself does not need a docstring.

Documenting functions and methods

The docstring for a function/method should start with a one-line description of what the function does, without referring to the function name or the names of variables. Use a "command style" for this summary, e.g. "Return the result." instead of "Returns the result." If the signature of the function cannot be automatically extracted because of an decorator (e.g. *check_units()*), place a signature in the very first row of the docstring, before the one-line description.

For methods, do not document the `self` parameter, nor give information about the method being static or a class method (this information will be automatically added to the documentation).

Documenting classes

Class docstrings should use the same "Parameters" and "Returns" sections as method and function docstrings for documenting the `__init__` constructor. If a class docstring does not have any "Attributes" or "Methods" section, these sections will be automatically generated with all documented (i.e. having a docstring), public (i.e. not starting with _) attributes respectively methods of the class. Alternatively, you can provide these sections manually. This is useful for example in the *Quantity* class, which would otherwise include the documentation of many ndarray methods, or when you want to include documentation for functions like `__getitem__` which would otherwise not be documented. When specifying these sections, you only have to state the names of documented methods/attributes but you can also provide direct documentation. For example:

```
Attributes
----------
foo
bar
baz
    This is a description.
```

This can be used for example for class or instance attributes which do not have "classical" docstrings. However, you can also use a special syntax: When defining class attributes in the class body or instance attributes in __init__ you can use the following variants (here shown for instance attributes):

```python
def __init__(a, b, c):
    #: The docstring for the instance attribute a.
    #: Can also span multiple lines
    self.a = a

    self.b = b #: The docstring for self.b (only one line).

    self.c = c
    'The docstring for self.c, directly *after* its definition'
```

Long example of a function docstring

This is a very long docstring, showing all the possible sections. Most of the time no See Also, Notes or References section is needed:

```python
def foo(var1, var2, long_var_name='hi') :
"""
A one-line summary that does not use variable names or the function name.

Several sentences providing an extended description. Refer to
variables using back-ticks, e.g. `var1`.

Parameters
----------
var1 : array_like
    Array_like means all those objects -- lists, nested lists, etc. --
    that can be converted to an array.  We can also refer to
    variables like `var1`.
var2 : int
    The type above can either refer to an actual Python type
    (e.g. ``int``), or describe the type of the variable in more
    detail, e.g. ``(N,) ndarray`` or ``array_like``.
Long_variable_name : {'hi', 'ho'}, optional
    Choices in brackets, default first when optional.


Returns
-------
describe : type
    Explanation
output : type
    Explanation
tuple : type
    Explanation
items : type
    even more explaining


Raises
------
BadException
    Because you shouldn't have done that.

See Also
```

(continues on next page)

```
--------
otherfunc : relationship (optional)
newfunc : Relationship (optional), which could be fairly long, in which
          case the line wraps here.
thirdfunc, fourthfunc, fifthfunc

Notes
-----
Notes about the implementation algorithm (if needed).

This can have multiple paragraphs.

You may include some math:

.. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}

And even use a greek symbol like :math:`omega` inline.

References
----------
Cite the relevant literature, e.g. [1]_.  You may also cite these
references in the notes section above.

.. [1] O. McNoleg, "The integration of GIS, remote sensing,
   expert systems and adaptive co-kriging for environmental habitat
   modelling of the Highland Haggis using object-oriented, fuzzy-logic
   and neural-network techniques," Computers & Geosciences, vol. 22,
   pp. 585-588, 1996.

Examples
--------
These are written in doctest format, and should illustrate how to
use the function.

>>> a=[1,2,3]
>>> print([x + 3 for x in a])
[4, 5, 6]
>>> print("a\nb")
a
b

"""

pass
```

7.1.6 Logging

For a description of logging from the users point of view, see *Logging*.

Logging in Brian is based on the `logging` module in Python's standard library.

Every brian module that needs logging should start with the following line, using the `get_logger()` function to get an instance of `BrianLogger`:

```
logger = get_logger(__name__)
```

In the code, logging can then be done via:

```
logger.diagnostic('A diagnostic message')
logger.debug('A debug message')
logger.info('An info message')
logger.warn('A warning message')
logger.error('An error message')
```

If a module logs similar messages in different places or if it might be useful to be able to suppress a subset of messages in a module, add an additional specifier to the logging command, specifying the class or function name, or a method name including the class name (do not include the module name, it will be automatically added as a prefix):

```
logger.debug('A debug message', 'CodeString')
logger.debug('A debug message', 'NeuronGroup.update')
logger.debug('A debug message', 'reinit')
```

If you want to log a message only once, e.g. in a function that is called repeatedly, set the optional `once` keyword to `True`:

```
logger.debug('Will only be shown once', once=True)
logger.debug('Will only be shown once', once=True)
```

The output of debugging looks like this in the log file:

```
2012-10-02 14:41:41,484 DEBUG    brian2.equations.equations.CodeString: A debug↵
↪message
```

and like this on the console (if the log level is set to "debug"):

```
DEBUG    A debug message [brian2.equations.equations.CodeString]
```

Log level recommendations

diagnostic Low-level messages that are not of any interest to the normal user but useful for debugging Brian itself. A typical example is the source code generated by the code generation module.

debug Messages that are possibly helpful for debugging the user's code. For example, this shows which objects were included in the network, which clocks the network uses and when simulations start and stop.

info Messages which are not strictly necessary, but are potentially helpful for the user. In particular, this will show messages about the chosen state updater and other information that might help the user to achieve better performance and/or accuracy in the simulations (e.g. using (event-driven) in synaptic equations, avoiding incompatible dt values between *TimedArray* and the *NeuronGroup* using it, …)

warn Messages that alert the user to a potential mistake in the code, e.g. two possible resolutions for an identifier in an equation. In such cases, the warning message should include clear information how to change the code to make the situation unambigous and therefore make the warning message disappear. It can also be used to make the user aware that he/she is using an experimental feature, an unsupported compiler or similar. In this case, normally the once=True option should be used to raise this warning only once. As a rule of thumb, "common" scripts like the examples provided in the examples folder should normally not lead to any warnings.

error This log level is not used currently in Brian, an exception should be raised instead. It might be useful in "meta-code", running scripts and catching any errors that occur.

The default log level shown to the user is info. As a general rule, all messages that the user sees in the default configuration (i.e., info and warn level) should be avoidable by simple changes in the user code, e.g. the renaming of variables, explicitly specifying a state updater instead of relying on the automatic system, adding (clock-driven)/(event-driven) to synaptic equations, etc.

Testing log messages

It is possible to test whether code emits an expected log message using the *catch_logs* context manager. This is normally not necessary for debug and info messages, but should be part of the unit tests for warning messages (*catch_logs* by default only catches warning and error messages):

```python
with catch_logs() as logs:
    # code that is expected to trigger a warning
    # ...
    assert len(logs) == 1
    # logs contains tuples of (log level, name, message)
    assert logs[0][0] == 'WARNING' and logs[0][1].endswith('warning_type')
```

7.1.7 Testing

Brian uses the pytest package for its testing framework.

Running the test suite

The pytest tool automatically finds tests in the code. However, to deal with the different code generation targets, and correctly set up tests for standalone mode, it is recommended to use Brian's builtin test function that calls pytest appropriately:

```python
>>> import brian2
>>> brian2.test()
```

By default, this runs the test suite for all available (runtime) code generation targets. If you only want to test a specific target, provide it as an argument:

```python
>>> brian2.test('numpy')
```

If you want to test several targets, use a list of targets:

```python
>>> brian2.test(['cython'])
```

In addition to the tests specific to a code generation target, the test suite will also run a set of independent tests (e.g. parsing of equations, unit system, utility functions, etc.). To exclude these tests, set the test_codegen_independent argument to False. Not all available tests are run by default, tests that take a long time are excluded. To include these, set long_tests to True.

To run the C++ standalone tests, you have to set the test_standalone argument to the name of a standalone device. If you provide an empty argument for the runtime code generation targets, you will only run the standalone tests:

```python
>>> brian2.test([], test_standalone='cpp_standalone')
```

Writing tests

Generally speaking, we aim for a 100% code coverage by the test suite. Less coverage means that some code paths are never executed so there's no way of knowing whether a code change broke something in that path.

Unit tests

The most basic tests are unit tests, tests that test one kind of functionality or feature. To write a new unit test, add a function called `test_...` to one of the `test_...` files in the `brian2.tests` package. Test files should roughly correspond to packages, test functions should roughly correspond to tests for one function/method/feature. In the test functions, use assertions that will raise an `AssertionError` when they are violated, e.g.:

```
G = NeuronGroup(42, model='dv/dt = -v / (10*ms) : 1')
assert len(G) == 42
```

When comparing arrays, use the `array_equal()` function from `numpy.testing.utils` which takes care of comparing types, shapes and content and gives a nicer error message in case the assertion fails. Never make tests depend on external factors like random numbers – tests should always give the same result when run on the same codebase. You should not only test the expected outcome for the correct use of functions and classes but also that errors are raised when expected. For that you can use pytest's `raises` function with which you can define a block of code that should raise an exception of a certain type:

```
with pytest.raises(DimensionMismatchError):
    3*volt + 5*second
```

You can also check whether expected warnings are raised, see the documentation of the *logging mechanism* for details

For simple functions, doctests (see below) are a great alternative to writing classical unit tests.

By default, all tests are executed for all selected runtime code generation targets (see *Running the test suite* above). This is not useful for all tests, some basic tests that for example test equation syntax or the use of physical units do not depend on code generation and need therefore not to be repeated. To execute such tests only once, they can be annotated with a `codegen_independent` marker, using the `mark` decorator:

```
import pytest
from brian2 import NeuronGroup

@pytest.mark.codegen_independent
def test_simple():
    # Test that the length of a NeuronGroup is correct
    group = NeuronGroup(5, '')
    assert len(group) == 5
```

Tests that are not "codegen-independent" are by default only executed for the runtimes device, i.e. not for the `cpp_standalone` device, for example. However, many of those tests follow a common pattern that is compatible with standalone devices as well: they set up a network, run it, and check the state of the network afterwards. Such tests can be marked as `standalone_compatible`, using the `mark` decorator in the same way as for `codegen_independent` tests.:

```
import pytest
from numpy.testing.utils import assert_equal
from brian2 import *

@pytest.mark.standalone_compatible
def test_simple_run():
```

(continues on next page)

```
    # Check that parameter values of a neuron don't change after a run
    group = NeuronGroup(5, 'v : volt')
    group.v = 'i*mV'
    run(1*ms)
    assert_equal(group.v[:], np.arange(5)*mV)
```

Tests that have more than a single run function but are otherwise compatible with standalone mode (e.g. they don't need access to the number of synapses or results of the simulation before the end of the simulation), can be marked as standalone_compatible and multiple_runs. They then have to use an explicit device.build(...) call of the form shown below:

```
import pytest
from numpy.testing.utils import assert_equal
from brian2 import *

@pytest.mark.standalone_compatible
@pytest.mark.multiple_runs
def test_multiple_runs():
    # Check that multiple runs advance the clock as expected
    group = NeuronGroup(5, 'v : volt')
    mon = StateMonitor(group, 'v', record=True)
    run(1 * ms)
    run(1 * ms)
    device.build(direct_call=False, **device.build_options)
    assert_equal(defaultclock.t, 2 * ms)
    assert_equal(mon.t[0], 0 * ms)
    assert_equal(mon.t[-1], 2 * ms - defaultclock.dt)
```

Tests can also be written specifically for a standalone device (they then have to include the *set_device* call and possibly the *build* call explicitly). In this case tests have to be annotated with the name of the device (e.g. 'cpp_standalone') and with 'standalone_only' to exclude this test from the runtime tests. Such code would look like this for a single *run()* call, i.e. using the automatic "build on run" feature:

```
import pytest
from brian2 import *

@pytest.mark.cpp_standalone
@pytest.mark.standalone_only
def test_cpp_standalone():
    set_device('cpp_standalone', directory=None)
    # set up simulation
    # run simulation
    run(...)
    # check simulation results
```

If the code uses more than one *run()* statement, it needs an explicit *build* call:

```
import pytest
from brian2 import *

@pytest.mark.cpp_standalone
@pytest.mark.standalone_only
def test_cpp_standalone():
    set_device('cpp_standalone', build_on_run=False)
    # set up simulation
    # run simulation
```

```
    run(...)
    # do something
    # run again
    run(...)
    device.build(directory=None)
    # check simulation results
```

Summary

@pytest.mark marker	Executed for devices	explicit use of *device*
codegen_independent	independent of devices	*none*
none	Runtime targets	*none*
standalone_compatible	Runtime and standalone	*none*
standalone_compatible, multiple_runs	Runtime and standalone	device.build(direct_call=False, **device.build_options)
cpp_standalone, standalone_only	C++ standalone device	set_device('cpp_standalone') ... device. build(directory=None)
my_device, standalone_only	"My device"	set_device('my_device') ... device. build(directory=None)

Doctests

Doctests are executable documentation. In the `Examples` block of a class or function documentation, simply write code copied from an interactive Python session (to do this from ipython, use `%doctestmode`), e.g.:

```
>>> from brian2.utils.stringtools import word_substitute
>>> expr = 'a*_b+c5+8+f(A)'
>>> print(word_substitute(expr, {'a':'banana', 'f':'func'}))
banana*_b+c5+8+func(A)
```

During testing, the actual output will be compared to the expected output and an error will be raised if they don't match. Note that this comparison is strict, e.g. trailing whitespace is not ignored. There are various ways of working around some problems that arise because of this expected exactness (e.g. the stacktrace of a raised exception will never be identical because it contains file names), see the doctest documentation for details.

Doctests can (and should) not only be used in docstrings, but also in the hand-written documentation, making sure that the examples actually work. To turn a code example into a doc test, use the `.. doctest::` directive, see *Equations* for examples written as doctests. For all doctests, everything that is available after `from brian2 import *` can be used directly. For everything else, add import statements to the doctest code or – if you do not want the import statements to appear in the document – add them in a `.. testsetup::` block. See the documentation for Sphinx's doctest extension for more details.

Doctests are a great way of testing things as they not only make sure that the code does what it is supposed to do but also that the documentation is up to date!

Correctness tests

[These do not exist yet for brian2]. Unit tests test a specific function or feature in isolation. In addition, we want to have tests where a complex piece of code (e.g. a complete simulation) is tested. Even if it is sometimes impossible to really check whether the result is correct (e.g. in the case of the spiking activity of a complex network), a useful check is also whether the result is *consistent*. For example, the spiking activity should be the same when using code generation for Python or C++. Or, a network could be pickled before running and then the result of the run could be compared to a second run that starts from the unpickled network.

7.2 Units

7.2.1 Casting rules

In Brian 1, a distinction is made between scalars and numpy arrays (including scalar arrays): Scalars could be multiplied with a unit, resulting in a Quantity object whereas the multiplication of an array with a unit resulted in a (unitless) array. Accordingly, scalars were considered as dimensionless quantities for the purpose of unit checking (e.g.. $1 + 1 * mV$ raised an error) whereas arrays were not (e.g. array(1) + 1 * mV resulted in 1.001 without any errors). Brian 2 no longer makes this distinction and treats both scalars and arrays as dimensionless for unit checking and make all operations involving quantities return a quantity.:

```
>>> 1 + 1*second
Traceback (most recent call last):
...
DimensionMismatchError: Cannot calculate 1. s + 1, units do not match (units are␣
↪second and 1).

>>> np.array([1]) + 1*second
Traceback (most recent call last):
...
DimensionMismatchError: Cannot calculate 1. s + [1], units do not match (units are␣
↪second and 1).

>>> 1*second + 1*second
2. * second
>>> np.array([1])*second + 1*second
array([ 2.]) * second
```

As one exception from this rule, a scalar or array `0` is considered as having "any unit", i.e. `0 + 1 * second` will result in `1 * second` without a dimension mismatch error and `0 == 0 * mV` will evaluate to `True`. This seems reasonable from a mathematical viewpoint and makes some sources of error disappear. For example, the Python builtin `sum` (not numpy's version) adds the value of the optional argument `start`, which defaults to 0, to its main argument. Without this exception, `sum([1 * mV, 2 * mV])` would therefore raise an error.

The above rules also apply to all comparisons (e.g. `==` or `<`) with one further exception: `inf` and `-inf` also have "any unit", therefore an expression like `v <= inf` will never raise an exception (and always return `True`).

7.2.2 Functions and units

ndarray methods

All methods that make sense on quantities should work, i.e. they check for the correct units of their arguments and return quantities with units were appropriate. Most of the methods are overwritten using thin function wrappers:

wrap_function_keep_dimension: Strips away the units before giving the array to the method of `ndarray`, then reattaches the unit to the result (examples: sum, mean, max)

wrap_function_change_dimension: Changes the dimensions in a simple way that is independent of function arguments, the shape of the array, etc. (examples: sqrt, var, power)

wrap_function_dimensionless: Raises an error if the method is called on a quantity with dimensions (i.e. it works on dimensionless quantities).

List of methods

`all`, `any`, `argmax`, `argsort`, `clip`, `compress`, `conj`, `conjugate`, `copy`, `cumsum`, `diagonal`, `dot`, `dump`, `dumps`, `fill`, `flatten`, `getfield`, `item`, `itemset`, `max`, `mean`, `min`, `newbyteorder`, `nonzero`, `prod`, `ptp`, `put`, `ravel`, `repeat`, `reshape`, `round`, `searchsorted`, `setasflat`, `setfield`, `setflags`, `sort`, `squeeze`, `std`, `sum`, `take`, `tolist`, `trace`, `transpose`, `var`, `view`

Notes

- Methods directly working on the internal data buffer (`setfield`, `getfield`, `newbyteorder`) ignore the dimensions of the quantity.

- The type of a quantity cannot be int, therefore `astype` does not quite work when trying to convert the array into integers.

- `choose` is only defined for integer arrays and therefore does not work

- `tostring` and `tofile` only return/save the pure array data without the unit (but you can use `dump` or `dumps` to pickle a quantity array)

- `resize` does not work: `ValueError: cannot resize this array: it does not own its data`

- `cumprod` would result in different dimensions for different elements and is therefore forbidden

- `item` returns a pure Python float by definition

- `itemset` does not check for units

Numpy ufuncs

All of the standard numpy ufuncs (functions that operate element-wise on numpy arrays) are supported, meaning that they check for correct units and return appropriate arrays. These functions are often called implicitly, for example when using operators like `<` or `**`.

Math operations: `add`, `subtract`, `multiply`, `divide`, `logaddexp`, `logaddexp2`, `true_divide`, `floor_divide`, `negative`, `power`, `remainder`, `mod`, `fmod`, `absolute`, `rint`, `sign`, `conj`, `conjugate`, `exp`, `exp2`, `log`, `log2`, `log10`, `expm1`, `log1p`, `sqrt`, `square`, `reciprocal`, `ones_like`

Trigonometric functions: `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `arctan2`, `hypot`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, `arctanh`, `deg2rad`, `rad2deg`

Bitwise functions: `bitwise_and`, `bitwise_or`, `bitwise_xor`, `invert`, `left_shift`, `right_shift`

Comparison functions: `greater`, `greater_equal`, `less`, `less_equal`, `not_equal`, `equal`, `logical_and`, `logical_or`, `logical_xor`, `logical_not`, `maximum`, `minimum`

Floating functions: `isreal, iscomplex, isfinite, isinf, isnan, floor, ceil, trunc, fmod`

Not taken care of yet: `signbit, copysign, nextafter, modf, ldexp, frexp`

Notes

- Everything involving `log` or `exp`, as well as trigonometric functions only works on dimensionless array (for `arctan2` and `hypot` this is questionable, though)

- Unit arrays can only be raised to a scalar power, not to an array of exponents as this would lead to differing dimensions across entries. For simplicity, this is enforced even for dimensionless quantities.

- Bitwise functions never works on quantities (numpy will by itself throw a `TypeError` because they are floats not integers).

- All comparisons only work for matching dimensions (with the exception of always allowing comparisons to 0) and return a pure boolean array.

- All logical functions treat quantities as boolean values in the same way as floats are treated as boolean: Any non-zero value is True.

Numpy functions

Many numpy functions are functional versions of ndarray methods (e.g. `mean`, `sum`, `clip`). They therefore work automatically when called on quantities, as numpy propagates the call to the respective method.

There are some functions in numpy that do not propagate their call to the corresponding method (because they use np.asarray instead of np.asanyarray, which might actually be a bug in numpy): `trace, diagonal, ravel, dot`. For these, wrapped functions in `unitsafefunctions.py` are provided.

Wrapped numpy functions in unitsafefunctions.py

These functions are thin wrappers around the numpy functions to correctly check for units and return quantities when appropriate:

`log, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh, diagonal, ravel, trace, dot`

numpy functions that work unchanged

This includes all functional counterparts of the methods mentioned above (with the exceptions mentioned above). Some other functions also work correctly, as they are only using functions/methods that work with quantities:

- `linspace, diff, digitize`[1]
- `trim_zeros, fliplr, flipud, roll, rot90, shuffle`
- `corrcoeff`[1]

numpy functions that return a pure numpy array instead of quantities

- `arange`
- `cov`
- `random.permutation`
- `histogram, histogram2d`
- `cross, inner, outer`
- `where`

numpy functions that do something wrong

---

[1] But does not care about the units of its input.

- `insert`, `delete` (return a quantity array but without units)

- `correlate` (returns a quantity with wrong units)

- `histogramdd` (raises a `DimensionMismatchError`)

other unsupported functions Functions in `numpy`'s subpackages such as `linalg` are not supported and will either not work with units, or remove units from their inputs.

User-defined functions and units

For performance and simplicity reasons, code within the Brian core does not use Quantity objects but unitless numpy arrays instead. See *Adding support for new functions* for details on how to make use user-defined functions with Brian's unit system.

7.3 Equations and namespaces

7.3.1 Equation parsing

Parsing is done via pyparsing, for now find the grammar at the top of the *brian2.equations.equations* file.

7.3.2 Variables

Each Brian object that saves state variables (e.g. *NeuronGroup*, *Synapses*, *StateMonitor*) has a `variables` attribute, a dictionary mapping variable names to `Variable` objects (in fact a `Variables` object, not a simple dictionary). `Variable` objects contain information *about* the variable (name, dtype, units) as well as access to the variable's value via a `get_value` method. Some will also allow setting the values via a corresponding `set_value` method. These objects can therefore act as proxies to the variables' "contents".

`Variable` objects provide the "abstract namespace" corresponding to a chunk of "abstract code", they are all that is needed to check for syntactic correctness, unit consistency, etc.

7.3.3 Namespaces

The `namespace` attribute of a group can contain information about the external (variable or function) names used in the equations. It specifies a group-specific namespace used for resolving names in that group. At run time, this namespace is combined with a "run namespace". This namespace is either explicitly provided to the *Network.run* method, or the implicit namespace consisting of the locals and globals around the point where the run function is called is used. This namespace is then passed down to all the objects via `Network.before_fun` which calls all the individual *BrianObject.before_run* methods with this namespace.

7.4 Variables and indices

7.4.1 Introduction

To be able to generate the proper code out of abstract code statements, the code generation process has to have access to information about the variables (their type, size, etc.) as well as to the indices that should be used for indexing arrays (e.g. a state variable of a `NeuronGroup` will be indexed differently in the `NeuronGroup` state updater and in synaptic propagation code). Most of this information is stored in the `variables` attribute of a `VariableOwner` (this includes `NeuronGroup`, `Synapses`, `PoissonGroup` and everything else that has state variables). The `variables` attribute can be accessed as a (read-only) dictionary, mapping variable names to `Variable` objects storing the information about the respective variable. However, it is not a simple dictionary but an instance of the `Variables` class. Let's have a look at its content for a simple example:

```
>>> tau = 10*ms
>>> G = NeuronGroup(10, 'dv/dt = -v / tau : volt')
>>> for name, var in sorted(G.variables.items()):
...     print('%s : %s' % (name, var))
...
N : <Constant(dimensions=Dimension(),  dtype=int64, scalar=True, constant=True, read_
→only=True)>
dt : <ArrayVariable(dimensions=second,  dtype=float, scalar=True, constant=True, read_
→only=True)>
i : <ArrayVariable(dimensions=Dimension(),  dtype=int32, scalar=False, constant=True,␣
→read_only=True)>
t : <ArrayVariable(dimensions=second,  dtype=float64, scalar=True, constant=False,␣
→read_only=True)>
t_in_timesteps : <ArrayVariable(dimensions=Dimension(),  dtype=int64, scalar=True,␣
→constant=False, read_only=True)>
v : <ArrayVariable(dimensions=metre ** 2 * kilogram * second ** -3 * amp ** -1, ␣
→dtype=float64, scalar=False, constant=False, read_only=False)>
```

The state variable `v` we specified for the `NeuronGroup` is represented as an `ArrayVariable`, all the other variables were added automatically. There's another array `i`, the neuronal indices (simply an array of integers from 0 to 9), that is used for string expressions involving neuronal indices. The constant `N` represents the total number of neurons. At the first sight it might be surprising that `t`, the current time of the clock and `dt`, its timestep, are `ArrayVariable` objects as well. This is because those values can change during a run (for `t`) or between runs (for `dt`), and storing them as arrays with a single value (note the `scalar=True`) is the easiest way to share this value – all code accessing it only needs a reference to the array and can access its only element.

The information stored in the `Variable` objects is used to do various checks on the level of the abstract code, i.e. before any programming language code is generated. Here are some examples of errors that are caught this way:

```
>>> G.v = 3*ms  # G.variables['v'].unit is volt
Traceback (most recent call last):
...
DimensionMismatchError: v should be set with a value with units volt, but got 3. ms␣
→(unit is second).
>>> G.N = 5  # G.variables['N'] is read-only
Traceback (most recent call last):
...
TypeError: Variable N is read-only
```

7.4.2 Creating variables

Each variable that should be accessible as a state variable and/or should be available for use in abstract code has to be created as a `Variable`. For this, first a `Variables` container with a reference to the group has to be created, individual variables can then be added using the various `add_...` methods:

```
self.variables = Variables(self)
self.variables.add_array('an_array', unit=volt, size=100)
self.variables.add_constant('N', unit=Unit(1), value=self._N, dtype=np.int32)
self.variables.create_clock_variables(self.clock)
```

As an additional argument, array variables can be specified with a specific *index* (see *Indices* below).

7.4.3 References

For each variable, only one `Variable` object exists even if it is used in different contexts. Let's consider the following example:

```
>>> G = NeuronGroup(5, 'dv/dt = -v / tau : volt', threshold='v > 1', reset='v = 0',
...                 name='neurons')
>>> subG = G[2:]
>>> S = Synapses(G, G, on_pre='v+=1*mV', name='synapses')
>>> S.connect()
```

All allow an access to the state variable `v` (note the different shapes, these arise from the different indices used, see below):

```
>>> G.v
<neurons.v: array([ 0.,  0.,  0.,  0.,  0.]) * volt>
>>> subG.v
<neurons_subgroup.v: array([ 0.,  0.,  0.]) * volt>
>>> S.v
<synapses.v: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]) * volt>
```

In all of these cases, the `Variables` object stores references to the same `ArrayVariable` object:

```
>>> id(G.variables['v'])
108610960
>>> id(subG.variables['v'])
108610960
>>> id(S.variables['v'])
108610960
```

Such a reference can be added using `Variables.add_reference`, note that the name used for the reference is not necessarily the same as in the original group, e.g. in the above example `S.variables` also stores references to `v` under the names `v_pre` and `v_post`.

7.4.4 Indices

In subgroups and especially in synapses, the transformation of abstract code into executable code is not straightforward because it can involve variables from different contexts. Here is a simple example:

```
>>> G = NeuronGroup(5, 'dv/dt = -v / tau : volt', threshold='v > 1', reset='v = 0')
>>> S = Synapses(G, G, 'w : volt', on_pre='v+=w')
```

The seemingly trivial operation `v+=w` involves the variable `v` of the *NeuronGroup* and the variable `w` of the *Synapses* object which have to be indexed in the appropriate way. Since this statement is executed in the context of `S`, the variable indices stored there are relevant:

```
>>> S.variables.indices['w']
'_idx'
>>> S.variables.indices['v']
'_postsynaptic_idx'
```

The index `_idx` has a special meaning and always refers to the "natural" index for a group (e.g. all neurons for a *NeuronGroup*, all synapses for a *Synapses* object, etc.). All other indices have to refer to existing arrays:

```
>>> S.variables['_postsynaptic_idx']
<DynamicArrayVariable(dimensions=Dimension(), dtype=<class 'numpy.int32'>,
↪scalar=False, constant=True, read_only=True)>
```

In this case, `_postsynaptic_idx` refers to a dynamic array that stores the postsynaptic targets for each synapse (since it is an array itself, it also has an index. It is defined for each synapse so its index is `_idx` – in fact there is currently no support for an additional level of indirection in Brian: a variable representing an index has to have `_idx` as its own index). Using this index information, the following C++ code (slightly simplified) is generated:

```cpp
for(int _spiking_synapse_idx=0;
    _spiking_synapse_idx<_num_spiking_synapses;
    _spiking_synapse_idx++)
{
    const int _idx = _spiking_synapses[_spiking_synapse_idx];
    const int _postsynaptic_idx = _ptr_array_synapses__synaptic_post[_idx];
    const double w = _ptr_array_synapses_w[_idx];
    double v = _ptr_array_neurongroup_v[_postsynaptic_idx];
    v += w;
    _ptr_array_neurongroup_v[_postsynaptic_idx] = v;
}
```

In this case, the "natural" index `_idx` iterates over all the synapses that received a spike (this is defined in the template) and `_postsynaptic_idx` refers to the postsynaptic targets for these synapses. The variables `w` and `v` are then pulled out of their respective arrays with these indices so that the statement `v += w;` does the right thing.

7.4.5 Getting and setting state variables

When a state variable is accessed (e.g. using `G.v`), the group does not return a reference to the underlying array itself but instead to a `VariableView` object. This is because a state variable can be accessed in different contexts and indexing it with a number/array (e.g. `obj.v[0]`) or a string (e.g. `obj.v['i>3']`) can refer to different values in the underlying array depending on whether the object is the *NeuronGroup*, a *Subgroup* or a *Synapses* object.

The `__setitem__` and `__getitem__` methods in `VariableView` delegate to `VariableView.set_item` and `VariableView.get_item` respectively (which can also be called directly under special circumstances). They analyze the arguments (is the index a number, a slice or a string? Is the target value an array or a string expression?) and delegate the actual retrieval/setting of the values to a specific method:

- Getting with a numerical (or slice) index (e.g. `G.v[0]`): `VariableView.get_with_index_array`

- Getting with a string index (e.g. `G.v['i>3']`): `VariableView.get_with_expression`

- Setting with a numerical (or slice) index and a numerical target value (e.g. `G.v[5:] = -70*mV`): `VariableView.set_with_index_array`

- Setting with a numerical (or slice) index and a string expression value (e.g. `G.v[5:] = (-70+i)*mV`): `VariableView.set_with_expression`

- Setting with a string index and a string expression value (e.g. `G.v['i>5'] = (-70+i)*mV`): `VariableView.set_with_expression_conditional`

These methods are annotated with the `device_override` decorator and can therefore be implemented in a different way in certain devices. The standalone device, for example, overrides the all the getting functions and the setting with index arrays. Note that for standalone devices, the "setter" methods do not actually set the values but only note them down for later code generation.

7.4.6 Additional variables and indices

The variables stored in the `variables` attribute of a *VariableOwner* can be used everywhere (e.g. in the state updater, in the threshold, the reset, etc.). Objects that depend on these variables, e.g. the `Thresholder` of a *NeuronGroup* add additional variables, in particular `AuxiliaryVariables` that are automatically added to the abstract code: a threshold condition `v > 1` is converted into the statement `_cond = v > 1`; to specify the meaning of the variable `_cond` for the code generation stage (in particular, C++ code generation needs to know the data type) an `AuxiliaryVariable` object is created.

In some rare cases, a specific `variable_indices` dictionary is provided that overrides the indices for variables stored in the `variables` attribute. This is necessary for synapse creation because the meaning of the variables changes in this context: an expression `v>0` does not refer to the `v` variable of all the *connected* postsynaptic variables, as it does under other circumstances in the context of a *Synapses* object, but to the `v` variable of all *possible* targets.

7.5 Preferences system

Each preference looks like `codegen.c.compiler`, i.e. dotted names. Each preference has to be registered and validated. The idea is that registering all preferences ensures that misspellings of a preference value by a user causes an error, e.g. if they wrote `codgen.c.compiler` it would raise an error. Validation means that the value is checked for validity, so `codegen.c.compiler = 'gcc'` would be allowed, but `codegen.c.compiler = 'hcc'` would cause an error.

An additional requirement is that the preferences system allows for extension modules to define their own preferences, including extending the existing core brian preferences. For example, an extension might want to define `extension.*` but it might also want to define a new language for codegen, e.g. `codegen.lisp.*`. However, extensions cannot add preferences to an existing category.

7.5.1 Accessing and setting preferences

Preferences can be accessed and set either keyword-based or attribute-based. To set/get the value for the preference example mentioned before, the following are equivalent:

```
prefs['codegen.c.compiler'] = 'gcc'
prefs.codegen.c.compiler = 'gcc'

if prefs['codegen.c.compiler'] == 'gcc':
    ...
if prefs.codegen.c.compiler == 'gcc':
    ...
```

Using the attribute-based form can be particulary useful for interactive work, e.g. in ipython, as it offers autocompletion and documentation. In ipython, `prefs.codegen.c?` would display a docstring with all the preferences available in the `codegen.c` category.

7.5.2 Preference files

Preferences are stored in a hierarchy of files, with the following order (each step overrides the values in the previous step but no error is raised if one is missing):

- The global defaults are stored in the installation directory.

- The user default are stored in `~/.brian/preferences` (which works on Windows as well as Linux).

- The file `brian_preferences` in the current directory.

7.5.3 Registration

Registration of preferences is performed by a call to *BrianGlobalPreferences.register_preferences*, e.g.:

```
register_preferences(
    'codegen.c',
    'Code generation preferences for the C language',
    'compiler'= BrianPreference(
        validator=is_compiler,
        docs='...',
        default='gcc'),
    ...
    )
```

The first argument `'codegen.c'` is the base name, and every preference of the form `codegen.c.*` has to be registered by this function (preferences in subcategories such as `codegen.c.somethingelse.*` have to be specified separately). In other words, by calling *register_preferences*, a module takes ownership of all the preferences with one particular base name. The second argument is a descriptive text explaining what this category is about. The preferences themselves are provided as keyword arguments, each set to a *BrianPreference* object.

7.5.4 Validation functions

A validation function takes a value for the preference and returns `True` (if the value is a valid value) or `False`. If no validation function is specified, a default validator is used that compares the value against the default value: Both should belong to the same class (e.g. int or str) and, in the case of a `Quantity` have the same unit.

7.5.5 Validation

Setting the value of a preference with a registered base name instantly triggers validation. Trying to set an unregistered preference using keyword or attribute access raises an error. The only exception from this rule is when the preferences are read from configuration files (see below). Since this happens before the user has the chance to import extensions that potentially define new preferences, this uses a special function (`_set_preference`). In this case,for base names that are not yet registered, validation occurs when the base name is registered. If, at the time that `Network.run` is called, there are unregistered preferences set, a `PreferenceError` is raised.

7.5.6 File format

The preference files are of the following form:

```
a.b.c = 1
# Comment line
[a]
b.d = 2
[a.b]
b.e = 3
```

This would set preferences `a.b.c=1`, `a.b.d=2` and `a.b.e=3`.

7.5.7 Built-in preferences

Brian itself defines the following preferences:

GSL

Directory containing GSL code

`GSL.directory = None` Set path to directory containing GSL header files (gsl_odeiv2.h etc.) If this directory is already in Python's include (e.g. because of conda installation), this path can be set to None.

codegen

Code generation preferences

`codegen.loop_invariant_optimisations = True`

Whether to pull out scalar expressions out of the statements, so that they are only evaluated once instead of once for every neuron/synapse/… Can be switched off, e.g. because it complicates the code (and the same optimisation is already performed by the compiler) or because the code generation target does not deal well with it. Defaults to `True`.

`codegen.max_cache_dir_size = 1000`

The size of a directory (in MB) with cached code for Cython that triggers a warning. Set to 0 to never get a warning.

`codegen.string_expression_target = 'numpy'`

Default target for the evaluation of string expressions (e.g. when indexing state variables). Should normally not be changed from the default numpy target, because the overhead of compiling code is not worth the speed gain for simple expressions.

Accepts the same arguments as *codegen.target*, except for `'auto'`

`codegen.target = 'auto'`

Default target for code generation.

Can be a string, in which case it should be one of:

- `'auto'` the default, automatically chose the best code generation target available.

- `'cython'`, uses the Cython package to generate C++ code. Needs a working installation of Cython and a C++ compiler.

- `'numpy'` works on all platforms and doesn't need a C compiler but is often less efficient.

Or it can be a `CodeObject` class.

codegen.cpp

C++ compilation preferences

`codegen.cpp.compiler = ''`

Compiler to use (uses default if empty). Should be `'unix'` or `'msvc'`.

To specify a specific compiler binary on unix systems, set the `CXX` environment variable instead.

`codegen.cpp.define_macros = []`

List of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or None to define it without a particular value (equivalent of "#define FOO" in source or -DFOO on Unix C compiler command line).

`codegen.cpp.extra_compile_args = None`

Extra arguments to pass to compiler (if None, use either `extra_compile_args_gcc` or `extra_compile_args_msvc`).

`codegen.cpp.extra_compile_args_gcc = ['-w', '-O3', '-ffast-math', '-fno-finite-math-only', '-march=native', '-std=c++11']`

Extra compile arguments to pass to GCC compiler

`codegen.cpp.extra_compile_args_msvc = ['/Ox', '/w', '', '/MP']`

Extra compile arguments to pass to MSVC compiler (the default `/arch:` flag is determined based on the processor architecture)

`codegen.cpp.extra_link_args = []`

Any extra platform- and compiler-specific information to use when linking object files together.

`codegen.cpp.headers = []`

A list of strings specifying header files to use when compiling the code. The list might look like ["<vector>","'my_header'"]. Note that the header strings need to be in a form than can be pasted at the end of a #include statement in the C++ code.

`codegen.cpp.include_dirs = ['/path/to/your/Python/environment/include']`

7.5. Preferences system

Include directories to use. The default value is $prefix/include (or $prefix/Library/ include on Windows), where $prefix is Python's site-specific directory prefix as returned by `sys. prefix`. This will make compilation use library files installed into a conda environment.

`codegen.cpp.libraries = []`

List of library names (not filenames or paths) to link against.

`codegen.cpp.library_dirs = ['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at link time. The default value is $prefix/lib (or $prefix/Library/lib on Windows), where $prefix is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

`codegen.cpp.msvc_architecture = ''`

MSVC architecture name (or use system architectue by default).

Could take values such as x86, amd64, etc.

`codegen.cpp.msvc_vars_location = ''`

Location of the MSVC command line tool (or search for best by default).

`codegen.cpp.runtime_library_dirs = ['/path/to/your/Python/environment/lib']`

List of directories to search for C/C++ libraries at run time. The default value is $prefix/lib (not used on Windows), where $prefix is Python's site-specific directory prefix as returned by `sys.prefix`. This will make compilation use library files installed into a conda environment.

codegen.generators

Codegen generator preferences (see subcategories for individual languages)

codegen.generators.cpp

C++ codegen preferences

`codegen.generators.cpp.flush_denormals = False`

Adds code to flush denormals to zero.

The code is gcc and architecture specific, so may not compile on all platforms. The code, for reference is:

```
#define CSR_FLUSH_TO_ZERO         (1 << 15)
unsigned csr = __builtin_ia32_stmxcsr();
csr |= CSR_FLUSH_TO_ZERO;
__builtin_ia32_ldmxcsr(csr);
```

Found at http://stackoverflow.com/questions/2487653/avoiding-denormal-values-in-c.

`codegen.generators.cpp.restrict_keyword = '__restrict'`

The keyword used for the given compiler to declare pointers as restricted.

This keyword is different on different compilers, the default works for gcc and MSVS.

codegen.runtime

Runtime codegen preferences (see subcategories for individual targets)

codegen.runtime.cython

Cython runtime codegen preferences

`codegen.runtime.cython.cache_dir = None`

Location of the cache directory for Cython files. By default, will be stored in a `brian_extensions` subdirectory where Cython inline stores its temporary files (the result of `get_cython_cache_dir()`).

`codegen.runtime.cython.delete_source_files` = `True`

Whether to delete source files after compiling. The Cython source files can take a significant amount of disk space, and are not used anymore when the compiled library file exists. They are therefore deleted by default, but keeping them around can be useful for debugging.

`codegen.runtime.cython.multiprocess_safe` = `True`

Whether to use a lock file to prevent simultaneous write access to cython .pyx and .so files.

codegen.runtime.numpy

Numpy runtime codegen preferences

`codegen.runtime.numpy.discard_units` = `False`

Whether to change the namespace of user-specifed functions to remove units.

core

Core Brian preferences

`core.default_float_dtype` = `float64`

Default dtype for all arrays of scalars (state variables, weights, etc.).

`core.default_integer_dtype` = `int32`

Default dtype for all arrays of integer scalars.

`core.outdated_dependency_error` = `True`

Whether to raise an error for outdated dependencies (`True`) or just a warning (`False`).

core.network

Network preferences

`core.network.default_schedule` = `['start', 'groups', 'thresholds', 'synapses', 'resets', 'end']`

Default schedule used for networks that don't specify a schedule.

devices

Device preferences

devices.cpp_standalone

C++ standalone preferences

`devices.cpp_standalone.extra_make_args_unix` = `['-j']`

Additional flags to pass to the GNU make command on Linux/OS-X. Defaults to "-j" for parallel compilation.

`devices.cpp_standalone.extra_make_args_windows` = `[]`

Additional flags to pass to the nmake command on Windows. By default, no additional flags are passed.

`devices.cpp_standalone.make_cmd_unix` = `'make'`

The make command used to compile the standalone project. Defaults to the standard GNU make commane "make".

`devices.cpp_standalone.openmp_spatialneuron_strategy` = `None`

DEPRECATED. Previously used to chose the strategy to parallelize the solution of the three tridiagonal systems for multicompartmental neurons. Now, its value is ignored.

`devices.cpp_standalone.openmp_threads` = `0`

The number of threads to use if OpenMP is turned on. By default, this value is set to 0 and the C++ code is generated without any reference to OpenMP. If greater than 0, then the corresponding number of threads are used to launch the simulation.

`devices.cpp_standalone.run_cmd_unix` = `'./main'`

The command used to run the compiled standalone project. Defaults to executing the compiled binary with "./main". Must be a single binary as string or a list of command arguments (e.g. ["./binary", "–key", "value"]).

`devices.cpp_standalone.run_environment_variables` = `{'LD_BIND_NOW': '1'}`

Dictionary of environment variables and their values that will be set during the execution of the standalone code.

legacy

Preferences to enable legacy behaviour

`legacy.refractory_timing` = `False`

Whether to use the semantics for checking the refractoriness condition that were in place up until (including) version 2.1.2. In that implementation, refractory periods that were multiples of dt could lead to a varying number of refractory timesteps due to the nature of floating point comparisons). This preference is only provided for exact reproducibility of previously obtained results, new simulations should use the improved mechanism which uses a more robust mechanism to convert refractoriness into timesteps. Defaults to `False`.

logging

Logging system preferences

`logging.console_log_level` = `'INFO'`

What log level to use for the log written to the console.

Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.delete_log_on_exit` = `True`

Whether to delete the log and script file on exit.

If set to `True` (the default), log files (and the copy of the main script) will be deleted after the brian process has exited, unless an uncaught exception occurred. If set to `False`, all log files will be kept.

`logging.display_brian_error_message` = `True`

Whether to display a text for uncaught errors, mentioning the location of the log file, the mailing list and the github issues.

Defaults to `True`.

`logging.file_log` = `True`

Whether to log to a file or not.

If set to `True` (the default), logging information will be written to a file. The log level can be set via the *logging.file_log_level* preference.

`logging.file_log_level = 'DIAGNOSTIC'`

What log level to use for the log written to the log file.

In case file logging is activated (see *logging.file_log*), which log level should be used for logging. Has to be one of CRITICAL, ERROR, WARNING, INFO, DEBUG or DIAGNOSTIC.

`logging.file_log_max_size = 10000000`

The maximum size for the debug log before it will be rotated.

If set to any value > 0, the debug log will be rotated once this size is reached. Rotating the log means that the old debug log will be moved into a file in the same directory but with suffix `".1"` and the a new log file will be created with the same pathname as the original file. Only one backup is kept; if a file with suffix `".1"` already exists when rotating, it will be overwritten. If set to 0, no log rotation will be applied. The default setting rotates the log file after 10MB.

`logging.save_script = True`

Whether to save a copy of the script that is run.

If set to `True` (the default), a copy of the currently run script is saved to a temporary location. It is deleted after a successful run (unless *logging.delete_log_on_exit* is `False`) but is kept after an uncaught exception occured. This can be helpful for debugging, in particular when several simulations are running in parallel.

`logging.std_redirection = True`

Whether or not to redirect stdout/stderr to null at certain places.

This silences a lot of annoying compiler output, but will also hide error messages making it harder to debug problems. You can always temporarily switch it off when debugging. If *logging.std_redirection_to_file* is set to `True` as well, then the output is saved to a file and if an error occurs the name of this file will be printed.

`logging.std_redirection_to_file = True`

Whether to redirect stdout/stderr to a file.

If both `logging.std_redirection` and this preference are set to `True`, all standard output/error (most importantly output from the compiler) will be stored in files and if an error occurs the name of this file will be printed. If *logging.std_redirection* is `True` and this preference is `False`, then all standard output/error will be completely suppressed, i.e. neither be displayed nor stored in a file.

The value of this preference is ignore if *logging.std_redirection* is set to `False`.

7.6 Adding support for new functions

For a description of Brian's function system from the user point of view, see *Functions*.

The default functions available in Brian are stored in the `DEFAULT_FUNCTIONS` dictionary. New *Function* objects can be added to this dictionary to make them available to all Brian code, independent of its namespace.

To add a new implementation for a code generation target, a `FunctionImplementation` can be added to the *Function.implementations* dictionary. The key for this dictionary has to be either a `CodeGenerator` class object, or a `CodeObject` class object. The `CodeGenerator` of a `CodeObject` (e.g. `CPPCodeGenerator` for `CPPStandaloneCodeObject`) is used as a fallback if no implementation specific to the `CodeObject` class exists.

If a function is already provided for the target language (e.g. it is part of a library imported by default), using the same name, all that is needed is to add an empty `FunctionImplementation` object to mark the function as implemented. For example, `exp` is a standard function in C++:

```
DEFAULT_FUNCTIONS['exp'].implementations[CPPCodeGenerator] = FunctionImplementation()
```

Some functions are implemented but have a different name in the target language. In this case, the `FunctionImplementation` object only has to specify the new name:

```
DEFAULT_FUNCTIONS['arcsin'].implementations[CPPCodeGenerator] =␣
→FunctionImplementation('asin')
```

Finally, the function might not exist in the target language at all, in this case the code for the function has to be provided, the exact form of this code is language-specific. In the case of C++, it's a dictionary of code blocks:

```
clip_code = {'support_code': '''
        double _clip(const float value, const float a_min, const float a_max)
        {
                if (value < a_min)
                    return a_min;
                if (value > a_max)
                    return a_max;
                return value;
        }
        '''}
DEFAULT_FUNCTIONS['clip'].implementations[CPPCodeGenerator] = FunctionImplementation(
→'_clip',
                                                                                    ␣
→code=clip_code)
```

7.7 Code generation

The generation of a code snippet is done by a `CodeGenerator` class. The templates are stored in the `CodeObject.templater` attribute, which is typically implemented as a subdirectory of templates. The compilation and running of code is done by a `CodeObject`. See the sections below for each of these.

7.7.1 Code path

The following gives an outline of the key steps that happen for the code generation associated to a *NeuronGroup* `StateUpdater`. The items in grey are Brian core functions and methods and do not need to be implemented to create a new code generation target or device. The parts in yellow are used when creating a new device. The parts in green relate to generating code snippets from abstract code blocks. The parts in blue relate to creating new templates which these snippets are inserted into. The parts in red relate to creating new runtime behaviour (compiling and running generated code).
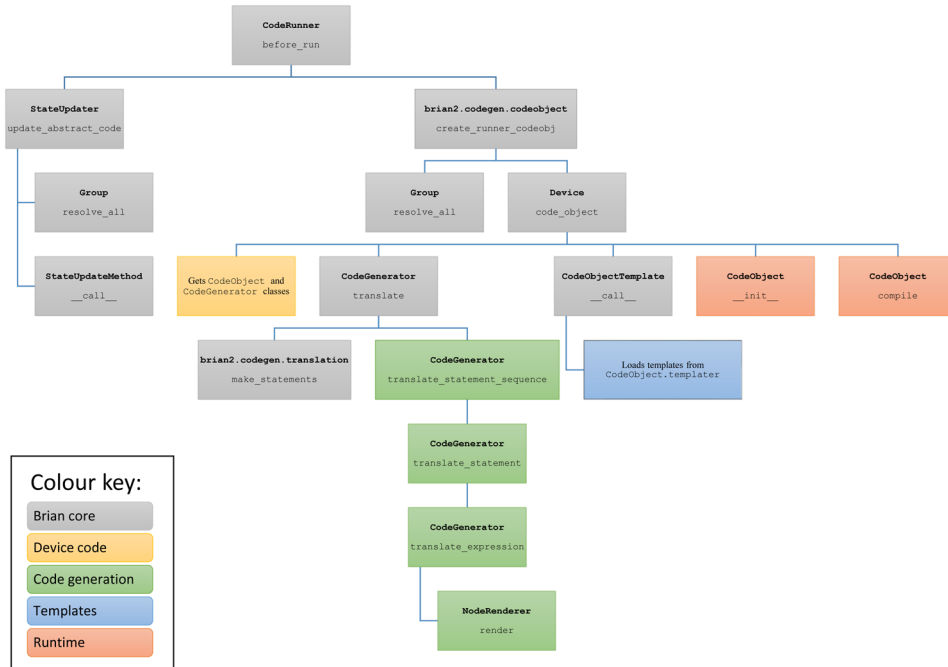
In brief, what happens can be summarised as follows. *Network.run* will call *BrianObject.before_run* on each of the objects in the network. Objects such as StateUpdater, which is a subclass of *CodeRunner* use this spot to generate and compile their code. The process for doing this is to first create the abstract code block, done in the StateUpdater.update_abstract_code method. Then, a CodeObject is created with this code block. In doing so, Brian will call out to the currently active Device to get the CodeObject and CodeGenerator classes associated to the device, and this hierarchy of calls gives several hooks which can be changed to implement new targets.

7.7.2 Code generation

To implement a new language, or variant of an existing language, derive a class from CodeGenerator. Good examples to look at are the NumpyCodeGenerator, CPPCodeGenerator and CythonCodeGenerator classes in the brian2.codegen.generators package. Each CodeGenerator has a class_name attribute which is a string used by the user to refer to this code generator (for example, when defining function implementations).

The derived CodeGenerator class should implement the methods marked as NotImplemented in the base Code-Generator class. CodeGenerator also has several handy utility methods to make it easier to write these, see the existing examples to get an idea of how these work.

7.7.3 Syntax translation

One aspect of writing a new language is that sometimes you need to translate from Python syntax into the syntax of another language. You are free to do this however you like, but we recommend using a NodeRenderer class which allows you to iterate over the abstract syntax tree of an expression. See examples in brian2.parsing.rendering.

7.7.4 Templates

In addition to snippet generation, you need to create templates for the new language. See the `templates` directories in `brian2.codegen.runtime.*` for examples of these. They are written in the Jinja2 templating system. The location of these templates is set as the `CodeObject.templater` attribute. Examples such as `CPPCodeObject` show how this is done.

Template structure

Languages typically define a `common_group` template that is the base for all other templates. This template sets up the basic code structure that will be reused by all code objects, e.g. by defining a function header and body, and adding standard imports/includes. This template defines several blocks, in particular a `maincode` clock containing the actual code that is specific to each code object. The specific templates such as `reset` then derive from the `common_group` base template and override the `maincode` block. The base template can also define additional blocks that are sometimes but not always overwritten. For example, the `common_group.cpp` template of the C++ standalone code generator defines an `extra_headers` block that can be overwritten by child templates to include additional header files needed for the code in `maincode`.

Template keywords

Templates also specify additional information necessary for the code generation process as Jinja comments (`{# ... #}`). The following keywords are recognized by Brian:

USES_VARIABLES Lists variable names that are used by the template, even if they are not referred to in user code.

WRITES_TO_READ_ONLY_VARIABLES Lists read-only variables that are modified by the template. Normally, read-only variables are not considered to change during code execution, but e.g. synapse creation requires changes to synaptic indices that are considered read-only otherwise.

ALLOWS_SCALAR_WRITE The presence of this keyword means that in this template, writing to scalar variables is permitted. Writing to scalar variables is not permitted by default, because it can be ambiguous in contexts that do not involve all neurons/synapses. For example, should the statement `scalar_variable += 1` in a reset statement update the variable once or once for every spiking neuron?

ITERATE_ALL Lists indices that are iterated over completely. For example, during the state update or threshold step, the template iterates over all neurons with the standard index `_idx`. When executing the reset statements on the other hand, not all neurons are concerned. This is only used for the numpy code generation target, where it allows avoiding expensive unnecessary indexing.

7.7.5 Code objects

To allow the final code block to be compiled and run, derive a class from `CodeObject`. This class should implement the placeholder methods defined in the base class. The class should also have attributes `templater` (which should be a `Templater` object pointing to the directory where the templates are stored) `generator_class` (which should be the `CodeGenerator` class), and `class_name` (which should be a string the user can use to refer to this code generation target.

7.7.6 Default functions

You will typically want to implement the default functions such as the trigonometric, exponential and `rand` functions. We usually put these implementations either in the same module as the `CodeGenerator` class or the `CodeObject` class depending on whether they are language-specific or runtime target specific. See those modules for examples of implementing these functions.

7.7.7 Code guide

- `brian2.codegen`: everything related to code generation

- `brian2.codegen.generators`: snippet generation, including the `CodeGenerator` classes and default function implementations.

- `brian2.codegen.runtime`: templates, compilation and running of code, including `CodeObject` and default function implementations.

- `brian2.core.functions`, `brian2.core.variables`: these define the values that variable names can have.

- `brian2.parsing`: tools for parsing expressions, etc.

- `brian2.parsing.rendering`: AST tools for rendering expressions in Python into different languages.

- `brian2.utils`: various tools for string manipulation, file management, etc.

7.7.8 Additional information

For some additional (older, but still accurate) notes on code generation:

Older notes on code generation

The following is an outline of how the Brian 2 code generation system works, with indicators as to which packages to look at and which bits of code to read for a clearer understanding.

We illustrate the global process with an example, the creation and running of a single *NeuronGroup* object:

- Parse the equations, add refractoriness to them: this isn't really part of code generation.

- Allocate memory for the state variables.

- Create `Thresholder`, `Resetter` and `StateUpdater` objects.
    - Determine all the variable and function names used in the respective abstract code blocks and templates
    - Determine the abstract namespace, i.e. determine a `Variable` or *Function* object for each name.
    - Create a `CodeObject` based on the abstract code, template and abstract namespace. This will generate code in the target language and the namespace in which the code will be executed.

- At runtime, each object calls `CodeObject.__call__` to execute the code.

Stages of code generation

Equations to abstract code

In the case of *Equations*, the set of equations are combined with a numerical integration method to generate an *abstract code block* (see below) which represents the integration code for a single time step.

An example of this would be converting the following equations:

```
eqs = '''
dv/dt = (v0-v)/tau : volt (unless refractory)
v0 : volt
'''
group = NeuronGroup(N, eqs, threshold='v>10*mV',
                    reset='v=0*mV', refractory=5*ms)
```

into the following abstract code using the *exponential_euler* method (which is selected automatically):

```
not_refractory = 1*((t - lastspike) > 0.005000)
_BA_v = -v0
_v = -_BA_v + (_BA_v + v)*exp(-dt*not_refractory/tau)
v = _v
```

The code for this stage can be seen in `NeuronGroup.__init__`, `StateUpdater.__init__`, and `StateUpdater.update_abstract_code` (in `brian2.groups.neurongroup`), and the *StateUpdateMethod* classes defined in the `brian2.stateupdaters` package.

For more details, see *State update*.

Abstract code

'Abstract code' is just a multi-line string representing a block of code which should be executed for each item (e.g. each neuron, each synapse). Each item is independent of the others in abstract code. This allows us to later generate code either for vectorised languages (like numpy in Python) or using loops (e.g. in C++).

Abstract code is parsed according to Python syntax, with certain language constructs excluded. For example, there cannot be any conditional or looping statements at the moment, although support for this is in principle possible and may be added later. Essentially, all that is allowed at the moment is a sequence of arithmetical `a = b*c` style statements.

Abstract code is provided directly by the user for threshold and reset statements in *NeuronGroup* and for pre/post spiking events in *Synapses*.

Abstract code to snippet

We convert abstract code into a 'snippet', which is a small segment of code which is syntactically correct in the target language, although it may not be runnable on its own (that's handled by insertion into a 'template' later). This is handled by the `CodeGenerator` object in `brian2.codegen.generators`. In the case of converting into python/numpy code this typically doesn't involve any changes to the code at all because the original code is in Python syntax. For conversion to C++, we have to do some syntactic transformations (e.g. `a**b` is converted to `pow(a, b)`), and add declarations for certain variables (e.g. converting `x=y*z` into `const double x = y*z;`).

An example of a snippet in C++ for the equations above:

```
const double v0 = _ptr_array_neurongroup_v0[_neuron_idx];
const double lastspike = _ptr_array_neurongroup_lastspike[_neuron_idx];
bool not_refractory = _ptr_array_neurongroup_not_refractory[_neuron_idx];
double v = _ptr_array_neurongroup_v[_neuron_idx];
not_refractory = 1 * (t - lastspike > 0.0050000000000000001);
const double _BA_v = -(v0);
const double _v = -(_BA_v) + (_BA_v + v) * exp(-(dt) * not_refractory / tau);
v = _v;
_ptr_array_neurongroup_not_refractory[_neuron_idx] = not_refractory;
_ptr_array_neurongroup_v[_neuron_idx] = v;
```

The code path that includes snippet generation will be discussed in more detail below, since it involves the concepts of namespaces and variables which we haven't covered yet.

Snippet to code block

The final stage in the generation of a runnable code block is the insertion of a snippet into a template. These use the Jinja2 template specification language. This is handled in `brian2.codegen.templates`.

An example of a template for Python thresholding:

```
# USES_VARIABLES { not_refractory, lastspike, t }
{% for line in code_lines %}
{{line}}
{% endfor %}
_return_values, = _cond.nonzero()
# Set the neuron to refractory
not_refractory[_return_values] = False
lastspike[_return_values] = t
```

and the output code from the example equations above:

```
# USES_VARIABLES { not_refractory, lastspike, t }
v = _array_neurongroup_v
_cond = v > 10 * mV
_return_values, = _cond.nonzero()
# Set the neuron to refractory
not_refractory[_return_values] = False
lastspike[_return_values] = t
```

Code block to executing code

A code block represents runnable code. Brian operates in two different regimes, either in runtime or standalone mode. In runtime mode, memory allocation and overall simulation control is handled by Python and numpy, and code objects operate on this memory when called directly by Brian. This is the typical way that Brian is used, and it allows for a rapid development cycle. However, we also support a standalone mode in which an entire project workspace is generated for a target language or device by Brian, which can then be compiled and run independently of Brian. Each mode has different templates, and does different things with the outputted code blocks. For runtime mode, in Python/numpy code is executed by simply calling the `exec` statement on the code block in a given namespace. In standalone mode, the templates will typically each be saved into different files.

Key concepts

Namespaces

In general, a namespace is simply a mapping/dict from names to values. In Brian we use the term 'namespace' in two ways: the high level "abstract namespace" maps names to objects based on the `Variables` or `Function` class. In the above example, `v` maps to an `ArrayVariable` object, `tau` to a `Constant` object, etc. This namespace has all the information that is needed for checking the consistency of units, to determine which variables are boolean or scalar, etc. During the `CodeObject` creation, this abstract namespace is converted into the final namespace in which the code will be executed. In this namespace, `v` maps to the numpy array storing the state variable values (without units) and `tau` maps to a concrete value (again, without units). See *Equations and namespaces* for more details.

Variable

`Variable` objects contain information about the variable they correspond to, including details like the data type, whether it is a single value or an array, etc.

See `brian2.core.variables` and, e.g. `Group._create_variables`, `NeuronGroup._create_variables`.

Templates

Templates are stored in Jinja2 format. They come in one of two forms, either they are a single template if code generation only needs to output a single block of code, or they define multiple Jinja macros, each of which is a separate code block. The `CodeObject` should define what type of template it wants, and the names of the macros to define. For examples, see the templates in the directories in `brian2/codegen/runtime`. See `brian2.codegen.templates` for more details.

Code guide

This section includes a guide to the various relevant packages and subpackages involved in the code generation process.

codegen Stores the majority of all code generation related code.

> **codegen.functions** Code related to including functions - built-in and user-defined - in generated code.
>
> **codegen.generators** Each `CodeGenerator` is defined in a module here.
>
> **codegen.runtime** Each runtime `CodeObject` and its templates are defined in a package here.

core

> **core.variables** The `Variable` types are defined here.

equations Everything related to *Equations*.

groups All *Group* related stuff is in here. The `Group.resolve` methods are responsible for determining the abstract namespace.

parsing Various tools using Python's `ast` module to parse user-specified code. Includes syntax translation to various languages in `parsing.rendering`.

stateupdaters Everything related to generating abstract code blocks from integration methods is here.

7.8 Devices

This document describes how to implement a new `Device` for Brian. This is a somewhat complicated process, and you should first be familiar with devices from the user point of view (*Computational methods and efficiency*) as well as the code generation system (*Code generation*).

We wrote Brian's devices system to allow for two major use cases, although it can potentially be extended beyond this. The two use cases are:

1. Runtime mode. In this mode, everything is managed by Python, including memory management (using numpy by default) and running the simulation. Actual computational work can be carried out in several different ways, including numpy or Cython.

2. Standalone mode. In this mode, running a Brian script leads to generating an entire source code project tree which can be compiled and run independently of Brian or Python.

Runtime mode is handled by `RuntimeDevice` and is already implemented, so here I will mainly discuss standalone devices. A good way to understand these devices is to look at the implementation of `CPPStandaloneDevice` (the only one implemented in the core of Brian). In many cases, the simplest way to implement a new standalone device would be to derive a class from `CPPStandaloneDevice` and overwrite just a few methods.

7.8.1 Memory management

Memory is managed primarily via the `Device.add_array`, `Device.get_value` and `Device.set_value` methods. When a new array is created, the `add_array` method is called, and when trying to access this memory the other two are called. The `RuntimeDevice` uses numpy to manage the memory and returns the underlying arrays in these methods. The `CPPStandaloneDevice` just stores a dictionary of array names but doesn't allocate any memory. This information is later used to generate code that will allocate the memory, etc.

7.8.2 Code objects

As in the case of runtime code generation, computational work is done by a collection of `CodeObject` s. In `CPPStandaloneDevice`, each code object is converted into a pair of `.cpp` and `.h` files, and this is probably a fairly typical way to do it.

7.8.3 Building

The method `Device.build` is used to generate the project. This can be implemented any way you like, although looking at `CPPStandaloneDevice.build` is probably a good way to get an idea of how to do it.

7.8.4 Device override methods

Several functions and methods in Brian are decorated with the `device_override` decorator. This mechanism allows a standalone device to override the behaviour of any of these functions by implementing a method with the name provided to `device_override`. For example, the `CPPStandaloneDevice` uses this to override *Network.run* as `CPPStandaloneDevice.network_run`.

7.8.5 Other methods

There are some other methods to implement, including initialising arrays, creating spike queues for synaptic propagation. Take a look at the source code for these.

7.9 Multi-threading with OpenMP

The following is an outline of how to make C++ standalone templates compatible with OpenMP, and therefore make them work in a multi-threaded environment. This should be considered as an extension to *Code generation*, that has to be read first. The C++ standalone mode of Brian is compatible with OpenMP, and therefore simulations can be launched by users with one or with multiple threads. Therefore, when adding new templates, the developers need to make sure that those templates are properly handling the situation if launched with OpenMP.

7.9.1 Key concepts

All the simulations performed with the C++ standalone mode can be launched with multi-threading, and make use of multiple cores on the same machine. Basically, all the Brian operations that can easily be performed in parallel, such as computing the equations for *NeuronGroup*, *Synapses*, and so on can and should be split among several threads. The network construction, so far, is still performed only by one single thread, and all created objects are shared by all the threads.

7.9.2 Use of `#pragma` flags

In OpenMP, all the parallelism is handled thanks to extra comments, added in the main C++ code, under the form:

```
#pragma omp ...
```

But to avoid any dependencies in the code that is generated by Brian when OpenMP is not activated, we are using functions that will only add those comments, during code generation, when such a multi-threading mode is turned on. By default, nothing will be inserted.

Translations of the `#pragma` commands

All the translations from `openmp_pragma()` calls in the C++ templates are handled in the file `devices/cpp_standalone/codeobject.py` In this function, you can see that all calls with various string inputs will generate #pragma statements inserted into the C++ templates during code generation. For example:

```
{{ openmp_pragma('static') }}
```

will be transformed, during code generation, into:

```
#pragma omp for schedule(static)
```

You can find the list of all the translations in the core of the `openmp_pragma()` function, and if some extra translations are needed, they should be added here.

Execution of the OpenMP code

In this section, we are explaining the main ideas behind the OpenMP mode of Brian, and how the simulation is executed in such a parallel context. As can be seen in `devices/cpp_standalone/templates/main.cpp`, the appropriate number of threads, defined by the user, is fixed at the beginning of the main function in the C++ code with:

```
{{ openmp_pragma('set_num_threads') }}
```

equivalent to (thanks to the `openmp_pragam()` function defined above): nothing if OpenMP is turned off (default), and to:

```
omp_set_dynamic(0);
omp_set_num_threads(nb_threads);
```

otherwise. When OpenMP creates a parallel context, this is the number of threads that will be used. As said, network creation is performed without any calls to OpenMP, on one single thread. Each template that wants to use parallelism has to add `{{ openmp_pragma{('parallel')}}` to create a general block that will be executed in parallel or `{{ openmp_pragma{('parallel-static')}}` to execute a single loop in parallel.

7.9.3 How to make your template use OpenMP parallelism

To design a parallel template, such as for example `devices/cpp_standalone/templates/common_group.cpp`, you can see that as soon as you have loops that can safely be split across nodes, you just need to add an openmp command in front of those loops:

```
{{openmp_pragma('parallel-static')}}
for(int _idx=0; _idx<N; _idx++)
{
    ...
}
```

By doing so, OpenMP will take care of splitting the indices and each thread will loop only on a subset of indices, sharing the load. By default, the scheduling use for splitting the indices is static, meaning that each node will get the same number of indices: this is the faster scheduling in OpenMP, and it makes sense for *NeuronGroup* or *Synapses* because operations are the same for all indices. By having a look at examples of templates such as `devices/cpp_standalone/templates/statemonitor.cpp`, you can see that you can merge portions of code executed by only one node and portions executed in parallel. In this template, for example, only one node is recording the time and extending the size of the arrays to store the recorded values:

```
{{_dynamic_t}}.push_back(_clock_t);

// Resize the dynamic arrays
{{_recorded}}.resize(_new_size, _num_indices);
```

But then, values are written in the arrays by all the nodes:

```
{{ openmp_pragma('parallel-static') }}
for (int _i = 0; _i < _num_indices; _i++)
{
    ....
}
```

In general, operations that manipulate global data structures, e.g. that use `push_back` for a `std::vector`, should only be executed by a single thread.

7.9.4 Synaptic propagation in parallel

General ideas

With OpenMP, synaptic propagation is also multi-threaded. Therefore, we have to modify the `SynapticPath-`
`way` objects, handling spike propagation. As can be seen in `devices/cpp_standalone/templates/`
`synapses_classes.cpp`, such an object, created during run time, will be able to get the number of threads decided
by the user:

```
_nb_threads = {{ openmp_pragma('get_num_threads') }};
```

By doing so, a `SynapticPathway`, instead of handling only one `SpikeQueue`, will be divided into `_nb_threads`
`SpikeQueues`, each of them handling a subset of the total number of connections. All the calls to `SynapticPathway`
object are performed from within `parallel` blocks in the `synapses` and `synapses_push_spikes` template,
we have to take this parallel context into account. This is why all the function of the `SynapticPathway` object are
taking care of the node number:

```
void push(int *spikes, unsigned int nspikes)
{
    queue[{{ openmp_pragma('get_thread_num') }}]->push(spikes, nspikes);
}
```

Such a method for the `SynapticPathway` will make sure that when spikes are propagated, all the threads will propa-
gate them to their connections. By default, again, if OpenMP is turned off, the queue vector has size 1.

Preparation of the `SynapticPathway`

Here we are explaining the implementation of the `prepare()` method for `SynapticPathway`:

```
{{ openmp_pragma('parallel') }}
{
    unsigned int length;
    if ({{ openmp_pragma('get_thread_num') }} == _nb_threads - 1)
        length = n_synapses - (unsigned int) {{ openmp_pragma('get_thread_num') }}*n_
→synapses/_nb_threads;
    else
        length = (unsigned int) n_synapses/_nb_threads;

    unsigned int padding  = {{ openmp_pragma('get_thread_num') }}*(n_synapses/_nb_
→threads);

    queue[{{ openmp_pragma('get_thread_num') }}]->openmp_padding = padding;
    queue[{{ openmp_pragma('get_thread_num') }}]->prepare(&real_delays[padding], &
→sources[padding], length, _dt);
}
```

Basically, each threads is getting an equal number of synapses (except the last one, that will get the remaining ones, if the
number is not a multiple of `n_threads`), and the queues are receiving a padding integer telling them what part of the
synapses belongs to each queue. After that, the parallel context is destroyed, and network creation can continue. Note
that this could have been done without a parallel context, in a sequential manner, but this is just speeding up everything.

Selection of the spikes

Here we are explaining the implementation of the `peek()` method for `SynapticPathway`. This is an example of concurrent access to data structures that are not well handled in parallel, such as `std::vector`. When `peek()` is called, we need to return a vector of all the neuron spiking at that particular time. Therefore, we need to ask every queue of the `SynapticPathway` what are the id of the spiking neurons, and concatenate them. Because those ids are stored in vectors with various shapes, we need to loop over nodes to perform this concatenate, in a sequential manner:

```
{{ openmp_pragma('static-ordered') }}
for(int _thread=0; _thread < {{ openmp_pragma('get_num_threads') }}; _thread++)
{
    {{ openmp_pragma('ordered') }}
    {
        if (_thread == 0)
            all_peek.clear();
        all_peek.insert(all_peek.end(), queue[_thread]->peek()->begin(), queue[_
→thread]->peek()->end());
    }
}
```

The loop, with the keyword 'static-ordered', is therefore performed such that node 0 enters it first, then node 1, and so on. Only one node at a time is executing the block statement. This is needed because vector manipulations can not be performed in a multi-threaded manner. At the end of the loop, `all_peek` is now a vector where all sub queues have written the id of spiking cells, and therefore this is the list of all spiking cells within the `SynapticPathway`.

7.9.5 Compilation of the code

One extra file needs to be modified, in order for OpenMP implementation to work. This is the makefile `devices/cpp_standalone/templates/makefile`. As one can simply see, the CFLAGS are dynamically modified during code generation thanks to:

```
{{ openmp_pragma('compilation') }}
```

If OpenMP is activated, this will add the following dependencies:

```
-fopenmp
```

such that if OpenMP is turned off, nothing, in the generated code, does depend on it.

7.10 Solving differential equations with the GNU Scientific Library

Conventionally, Brian generates its own code performing *Numerical integration* according to the chosen algorithm (see the section on *Code generation*). Another option is to let the differential equation solvers defined in the GNU Scientific Library (GSL) solve the given equations. In addition to offering a few extra integration methods, the GSL integrator comes with the option of having an adaptable timestep. The latter functionality can have benefits for the speed with which large simulations can be run. This is because it allows the use of larger timesteps for the overhead loops in Python, without losing the accuracy of the numerical integration at points where small timesteps are necessary. In addition, a major benefit of using the ODE solvers from GSL is that an estimation is performed on how wrong the current solution is, so that simulations can be performed with some confidence on accuracy. (Note however that the confidence of accuracy is based on estimation!)

7.10.1 StateUpdateMethod

Translation of equations to abstract code

The first part of Brian's code generation is the translation of equations to what we call 'abstract code'. In the case of Brian's stateupdaters so far, this abstract code describes the calculations that need to be done to update differential variables depending on their equations as is explained in the section on *State update*. In the case of preparing the equations for GSL integration this is a bit different. Instead of writing down the computations that have to be done to reach the new value of the variable after a time step, the equations have to be described in a way that GSL understands. The differential equations have to be defined in a function and the function is given to GSL. This is best explained with an example. If we have the following equations (taken from the adaptive threshold example):

```
dv/dt = -v/(10*ms) : volt
dvt/dt = (10*mV - vt)/(15*ms) : volt
```

We would describe the equations to GSL as follows:

```
v = y[0]
vt = y[1]
f[0] = -v/(10e-3)
f[1] = (10e-3 - vt)
```

Each differential variable gets an index. Its value at any time is saved in the `y`-array and the derivatives are saved in the `f`-array. However, doing this translation in the stateupdater would mean that Brian has to deal with variable descriptions that contain array accessing: something that for example sympy doesn't do. Because we still want to use Brian's existing parsing and checking mechanisms, we needed to find a way to describe the abstract code with only 'normal' variable names. Our solution is to replace the `y[0]`, `f[0]`, etc. with a 'normal' variable name that is later replaced just before the final code generation (in the `GSLCodeGenerator`). It has a tag and all the information needed to write the final code. As an example, the GSL abstract code for the above equations would be:

```
v = _gsl_y0
vt = _gsl_y1
_gsl_f0 = -v/(10e-3)
_gsl_f1 = (10e-3 - vt)
```

In the `GSLCodeGenerator` these tags get replaced by the actual accessing of the arrays.

Return value of the StateUpdateMethod

So far, for each each code generation language (numpy, cython) there was just one set of rules of how to translate abstract code to real code, described in its respective `CodeObject` and `CodeGenerator`. If the target language is set to Cython, the stateupdater will use the *CythonCodeObject*, just like other objects such as the *StateMonitor*. However, to achieve the above decribed translations of the abstract code generated by the *StateUpdateMethod*, we need a special *CythonCodeObject* for the stateupdater alone (which at its turn can contain the special `CodeGenerator`), and this `CodeObject` should be selected based on the chosen *StateUpdateMethod*.

In order to achieve `CodeObject` selection based on the chosen stateupdater, the *StateUpdateMethod* returns a class that can be called with an object, and the appropriate `CodeObject` is added as an attribute to the given object. The return value of this callable is the abstract code describing the equations in a language that makes sense to the `GSLCodeGenerator`.

7.10.2 GSLCodeObject

Each target language has its own `GSLCodeObject` that is derived from the already existing code object of its language. There are only minimal changes to the already existing code object:

- Overwrite `stateupate` template: a new version of the `stateupdate` template is given (`stateupdate.cpp` for C++ standalone and `stateupdate.pyx` for cython).

- Have a GSL specific generator_class: `GSLCythonCodeGenerator`

- Add the attribute `original_generator_class`: the conventional target-language generator is used to do the bulk of the translation to get from abstract code to language-specific code.

This defining of GSL-specific code objects also allowed us to catch compilation errors so we can give the user some information on that it might be GSL-related (overwriting the `compile()` method in the case of cython). In the case of the C++ `CodeObject` such overriding wasn't really possible so compilation errors in this case might be quite undescriptive.

7.10.3 GSLCodeGenerator

This is where the magic happens. Roughly 1000 lines of code define the translation of abstract code to code that uses the GNU Scientific Library's ODE solvers to achieve state updates.

Upon a call to *`run()`*, the code objects necessary for the simulation get made. The code for this is described in the device. Part of making the code objects is generating the code that describes the code objects. This starts with a call to `translate`, which in the case of GSL brings us to the `GSLCodeGenerator.translate()`. This method is built up as follows:

- Some GSL-specific preparatory work:

  - Check whether the equations contain variable names that are reserved for the GSL code.

  - Add the 'gsl tags' (see section on StateUpdateMethod) to the variables known to Brian as non-scalars. This is necessary to ensure that all equations containing 'gsl tags' are considered vector equations, and thus added to Brian's vector code.

  - Add GSL integrator meta variables as official Brian variables, so these are also taken into account upon translation. The meta variables that are possible are described in the user manual (e.g. GSL's step taken in a single overhead step '_step_count').

  - Save function names. The original generators delete the function names from the variables dictionary once they are processed. However, we need to know later in the GSL part of the code generation whether a certain encountered variable name refers to a function or not.

- Brian's general preparatory work. This piece of code is directly copied from the base CodeGenerator and is thus similar to what is done normally.

- A call to `original_generator.translate()` to get the abstract code translated into code that is target-language specific.

- A lot of statements to translate the target-language specific code to GSL-target-language specific code, described in more detail below.

The biggest difference between conventional Brian code and GSL code is that the stateupdate-decribing lines are contained directly in the `main()` or in a separate function, respectively. In both cases, the equations describing the system refer to parameters that are in the Brian namespace (e.g. "dv/dt = -v/tau" needs access to "tau"). How can we access Brian's namespace in this separate function that is needed with GSL?

To explain the solution we first need some background information on this 'separate function' that is given to the GSL integrators: `_GSL_func`. This function always gets three arguments:

- `double t`: the current time. This is relevant when the equations are dependent on time.

- `const double _GSL_y[]`': an array containing the current values of the differential variables (const because the cannot be changed by _GSL_func itself).

- `double f[]`: an array containing the derivatives of the differential variables (i.e. the equations describing the differential system).

- `void * params`: a pointer.

The pointer can be a pointer to whatever you want, and can thus point to a data structure containing the system parameters (such as tau). To achieve a structure containing all the parameters of the system, a considerable amount of code has to be added/changed to that generated by conventional Brian:

- The data structure, _GSL_dataholder, has to be defined with all variables needed in the vector code. For this reason, also the datatype of each variable is required.

    - This is done in the method `GSLCodeGenerator.write_dataholder`

- Instead of referring to the variables by their name only (e.g. `dv/dt = -v/tau`), the variables have to be accessed as part of the data structure (e.g. `dv/dt = -v/_GSL_dataholder->tau` in the case of cpp). Also, as mentioned earlier, we want to translate the 'gsl tags' to what they should be in the final code (e.g. `_gsl_f0` to `f[0]`).

    - This is done in the method `GSLCodeGenerator.translate_vector_code`. It works based on the to_replace dictionary (generated in the methods `GSLCodeGenerator.diff_var_to_replace` and `GSLCodeGenerator.to_replace_vector_vars`) that simply contains the old variables as keys and new variables as values, and is given to the word_replace function.

- The values of the variables in the data structure have to be set to the values of the variables in the Brian namespace.

    - This is done in the method `GSLCodeGenerator.unpack_namespace`, and for the 'scalar' variables that require calculation first it is done in the method `GSLCodeGenerator.translate_scalar_code`.

In addition, a few more 'support' functions are generated for the GSL script:

- `int _set_dimension(size_t * dimension)`: sets the dimension of the system. Required for GSL.

- `double* _assign_memory_y()`: allocates the right amount of memory for the y array (also according to the dimension of the system).

- `int _fill_y_vector(_dataholder* _GSL_dataholder, double* _GSL_y, int _idx)`: pulls out the values for each differential variable out of the 'Brian' array into the y-vector. This happens in the vector loop (e.g. `y[0] = _GSL_dataholder->_ptr_array_neurongroup_v[_idx]`; for C++).

- `int _empty_y_vector(_dataholder* _GSL_dataholder, double* _GSL_y, int _idx)`: the opposite of _fill_y_vector. Pulls final numerical solutions from the y array and gives it back to Brian's namespace.

- `double* _set_GSL_scale_array()`: sets the array bound for each differential variable, for which the values are based on `method_options['absolute_error']` and `method_options['absolute_error_per_variable']`.

All of this is written in support functions so that the vector code in the `main()` can stay almost constant for any simulation.

7.10.4 Stateupdate templates

There is many extra things that need to be done for each simulation when using GSL compared to conventional Brian stateupdaters. These are summarized in this section.

Things that need to be done for every type of simulation (either before, in or after `main()`):

- Cython-only: define the structs and functions that we will be using in cython language.

- Prepare the `gsl_odeiv2_system`: give function pointer, set dimension, give pointer to `_GSL_dataholder` as params.

- Allocate the driver (name for the struct that contains the info necessary to perform GSL integration)

- Define dt.

Things that need to be done every loop iteration for every type of simulation:

- Define t and t1 (t + dt).

- Transfer the values in the Brian arrays to the y-array that will be given to GSL.

- Set `_GSL_dataholder._idx` (in case we need to access array variables in `_GSL_func`).

- Initialize the driver (reset counters, set `dt_start`).

- Apply driver (either with adaptable- or fixed time step).

- Optionally save certain meta-variables

- Transfer values from GSL's y-vector to Brian arrays

INDICES AND TABLES

- genindex

- modindex

- search

[R13] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R14] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arccosh

[R15] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R16] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arcsinh

[R17] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[R18] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arctanh

[R19] Wikipedia, "Exponential function", https://en.wikipedia.org/wiki/Exponential_function

[R20] M. Abramovitz and I. A. Stegun, "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables," Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

[R21] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[R22] Wikipedia, "Logarithm". https://en.wikipedia.org/wiki/Logarithm

[R23] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[R24] Wikipedia, "Hyperbolic function", https://en.wikipedia.org/wiki/Hyperbolic_function

PYTHON MODULE INDEX

M

N

X

Y

Z